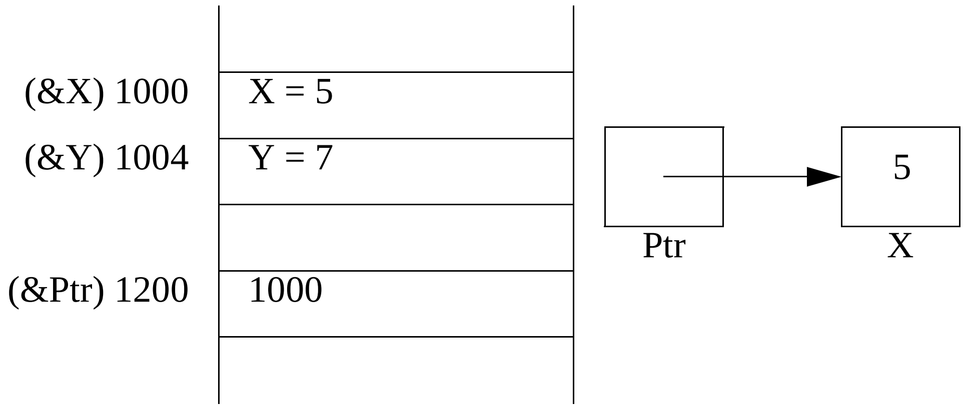
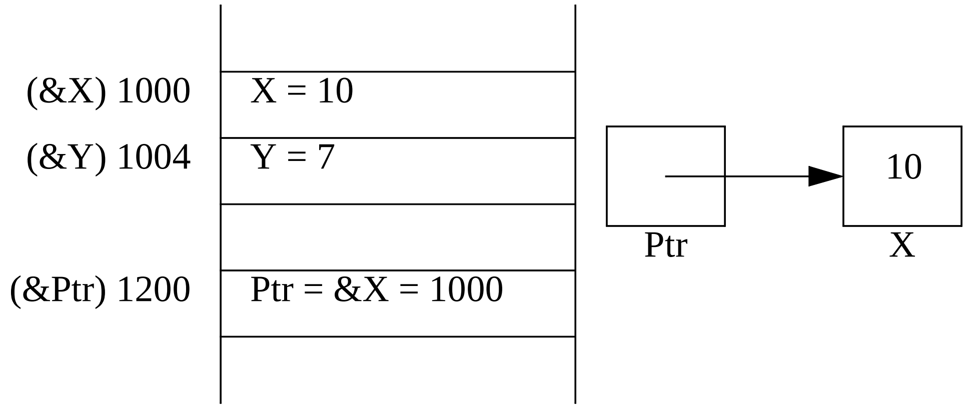
Chapter 1

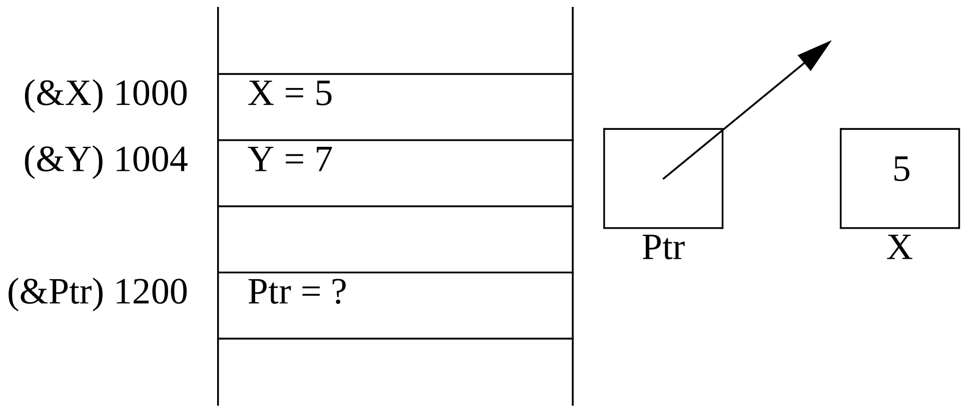
Pointers, Arrays, and Structures



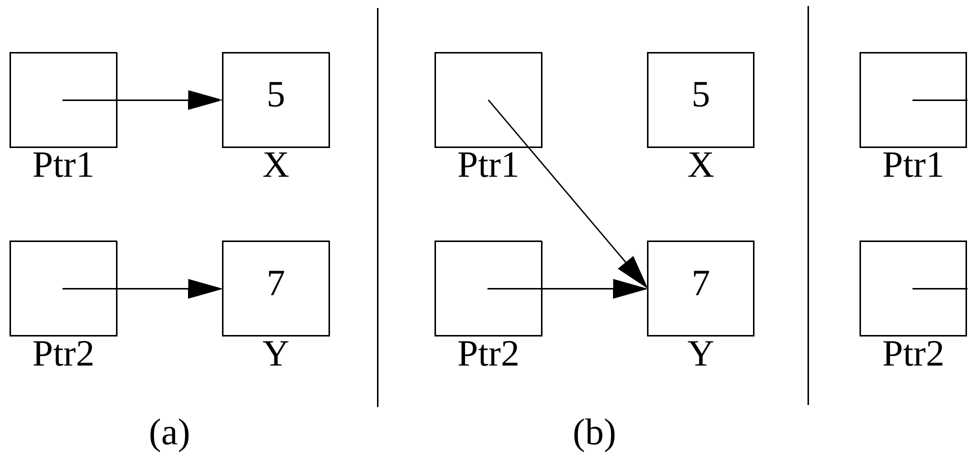
Pointer illustration



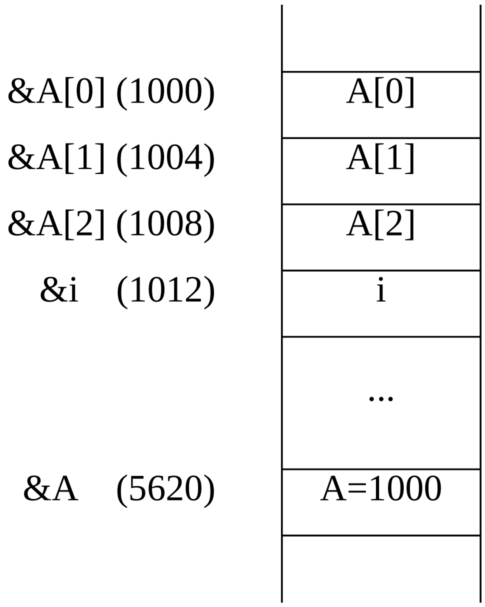
Result of `*Ptr=10`



Uninitialized pointer



(a) Initial state; (b) `Ptr1=Ptr2` starting from initial state;
(c) `*Ptr1=*Ptr2` starting from initial state



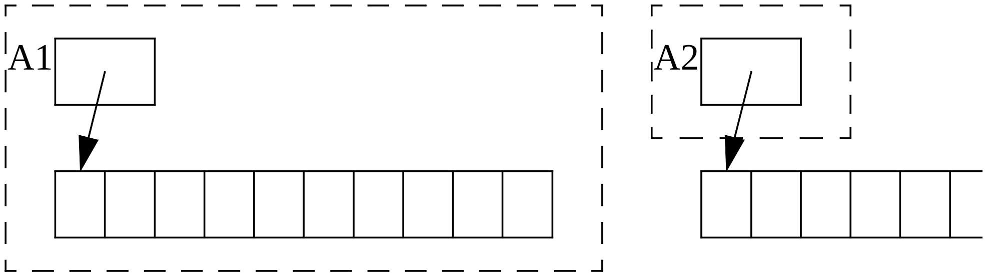
Memory model for arrays (assumes 4 byte `int`); declaration is `int A[3]; int i;`


```
size_t strlen( const char *Str );
char * strcpy(       char *Lhs, const char *Rhs );
char * strcat(       char *Lhs, const char *Rhs );
int    strcmp( const char *Lhs, const char *Rhs );
```

Some of the string routines in <string.h>

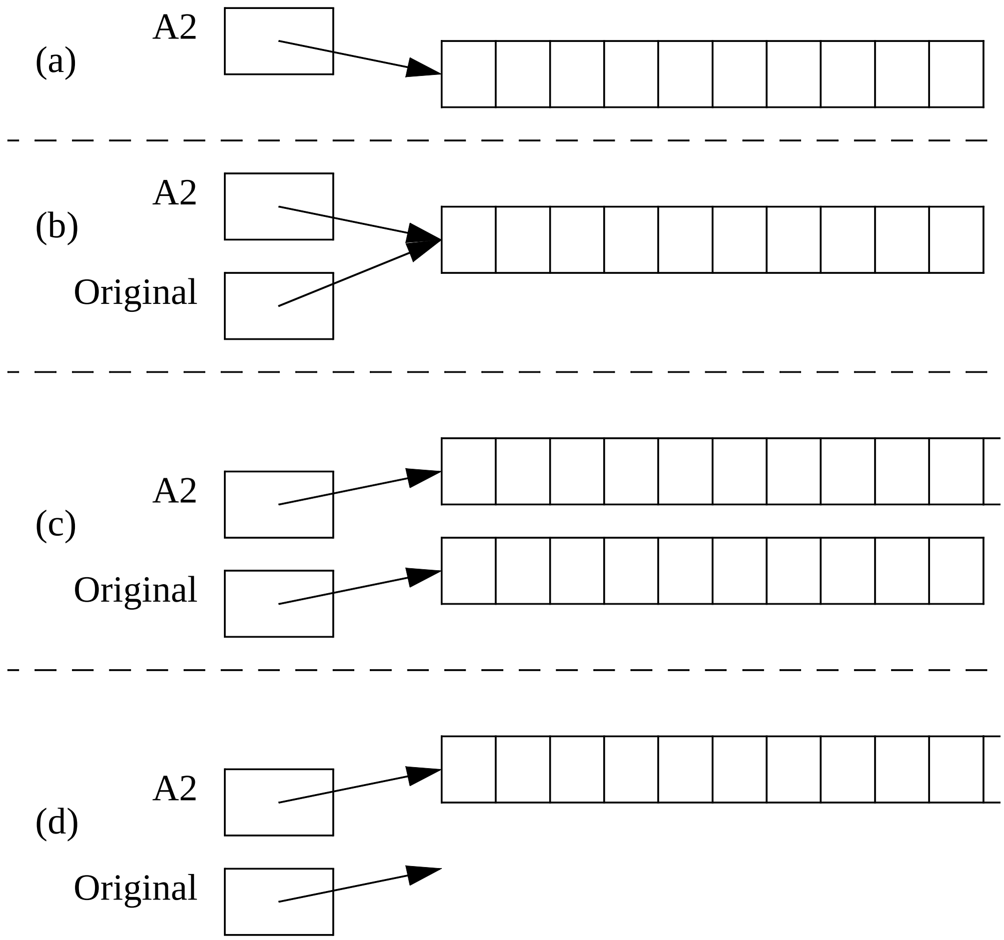

```
void
F( int i )
{
    int A1[ 10 ];
    int *A2 = new int [ 10 ];

    ...
    G( A1 );
    G( A2 );

    // On return, all memory associated with A1 is freed
    // On return, only the pointer A2 is freed;
    // 10 ints have leaked
    // delete [ ] A2;   // This would fix the leak
}
```

Two ways to allocate arrays; one leaks memory

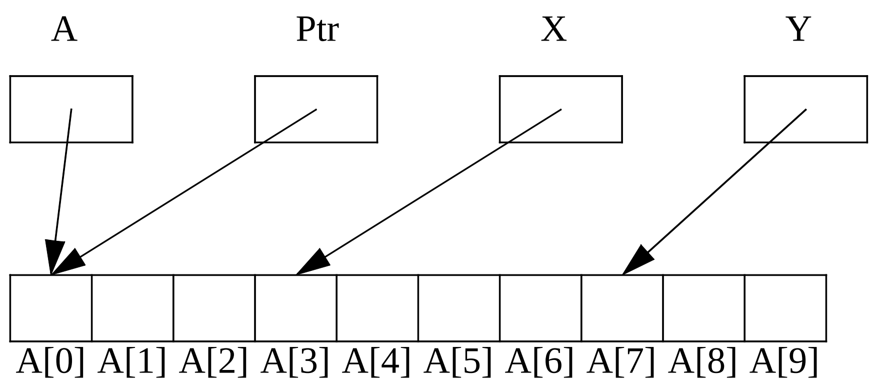



```
int *Original = A2;       // 1. Save pointer to the original
A2 = new int [ 12 ];      // 2. Have A2 point at more memory
for( int i = 0; i < 10; i++ ) // 3. Copy the old data over
    A2[ i ] = Original[ i ];
delete [ ] Original;      // 4. Recycle the original array
```

Memory reclamation



Array expansion: (a) starting point: `A2` points at 10 integers; (b) after step 1: `Original` points at the 10 integers; (c) after steps 2 and 3: `A2` points at 12 integers, the first 10 of which are copied from `Original`; (d) after step 4: the 10 integers are freed



Pointer arithmetic: `X=&A[3]; Y=X+4`


```
// Test that Strlen1 and Strlen2 give same answer
// Source file is ShowProf.cpp

#include <iostream.h>

main( )
{
    char Str[ 512 ];

    while( cin >> Str )
    {
        if( Strlen1( Str ) != Strlen2( Str ) )
            cerr << "Oops!!!!" << endl;
    }

    return 0;
}
```

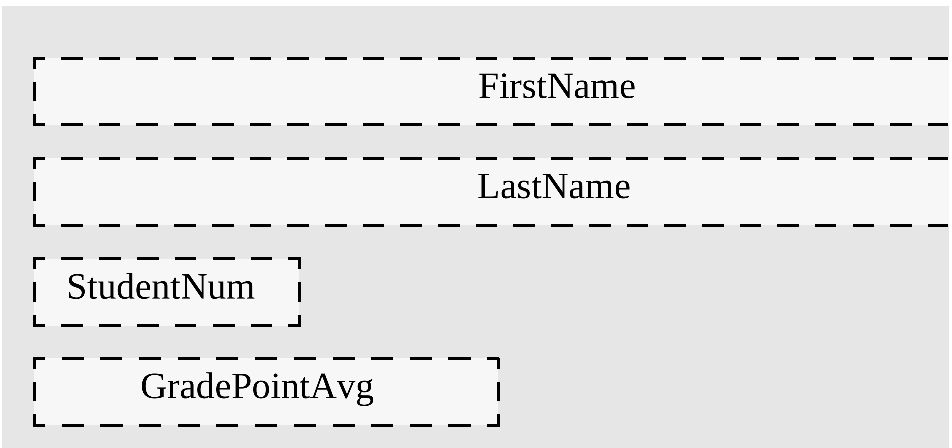
%time	cumsecs	#call	ms/call	name
26.6	0.34	25145	0.01	___rs__7istreamFPc
22.7	0.63	25144	0.01	_Strlen2__FPCc
14.8	0.82			mcount
12.5	0.98	25144	0.01	_Strlen1__FPCc
8.6	1.09	25145	0.00	_do_ipfx__7istreamFi
6.2	1.17	25145	0.00	_eatwhite__7istreamFv
4.7	1.23	204	0.29	_read
3.1	1.27	1	40.00	_main

First eight lines from `prof` for program

%time	cumsecs	#call	ms/call	name
34.4	0.31			mcount
26.7	0.55	25145	0.01	___rs__7istreamFPc
8.9	0.63	25145	0.00	_do_ipfx__7istreamFi
6.7	0.69	25144	0.00	_Strlen1__FPCc
6.7	0.75	25144	0.00	_Strlen2__FPCc
6.7	0.81	25145	0.00	_eatwhite__7istreamFv
6.7	0.87	204	0.29	_read
3.3	0.90	1	30.00	_main

First eight lines from `prof` with highest optimization

```
struct Student
{
    char FirstName[ 40 ];
    char LastName[ 40 ];
    int StudentNum;
    double GradePointAvg;
};
```



`Student` structure

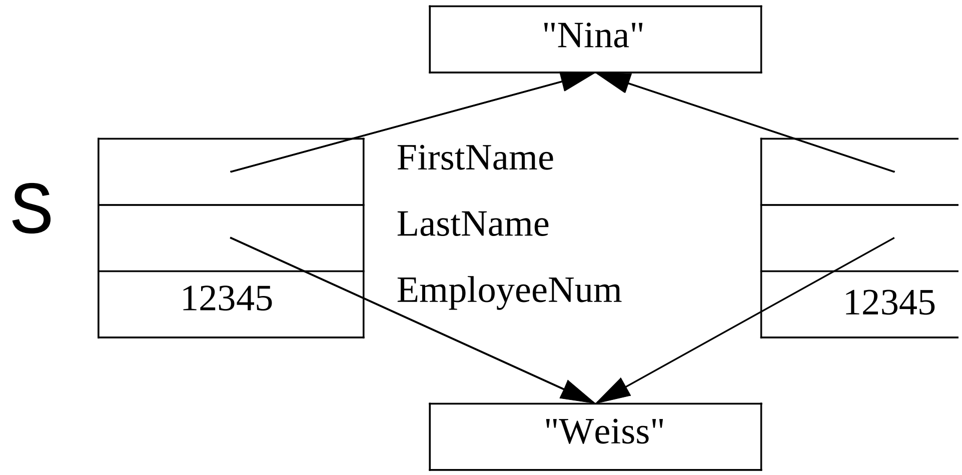


Illustration of a shallow copy in which only pointers are copied

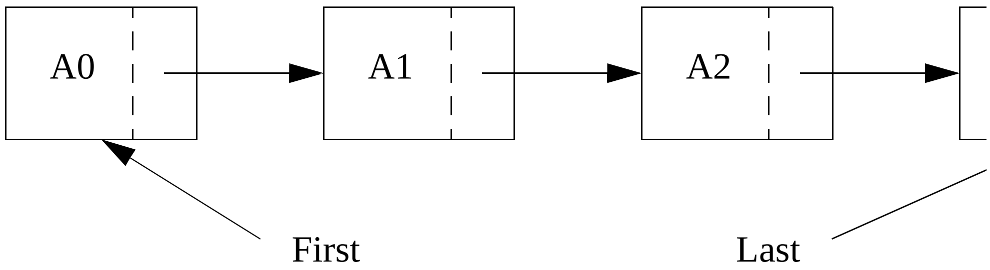


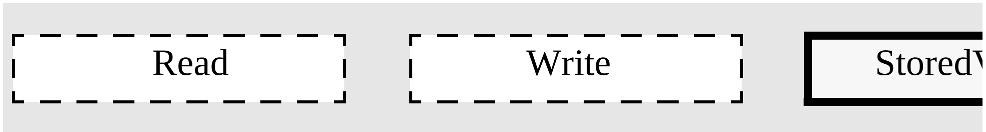
Illustration of a simple linked list

Chapter 2

Objects and Classes


```
// MemoryCell class
//  int Read( )         -->  Returns the stored value
//  void Write( int X ) -->  X is stored

class MemoryCell
{
  public:
        // Public member functions
    int Read( )           { return StoredValue; }
    void Write( int X )   { StoredValue = X; }
  private:
        // Private internal data representation
    int StoredValue;
};
```

A complete declaration of a `MemoryCell` class



`MemoryCell` members: `Read` and `Write` are accessible, but `StoredValue` is hidden


```
// Exercise the MemoryCell class

main( )
{
    MemoryCell M;

    M.Write( 5 );
    cout << "Cell contents are " << M.Read( ) << '\n';
        // The next line would be illegal if uncommented
//  cout << "Cell contents are " << M.StoredValue << '\n';
    return 0;
}
```

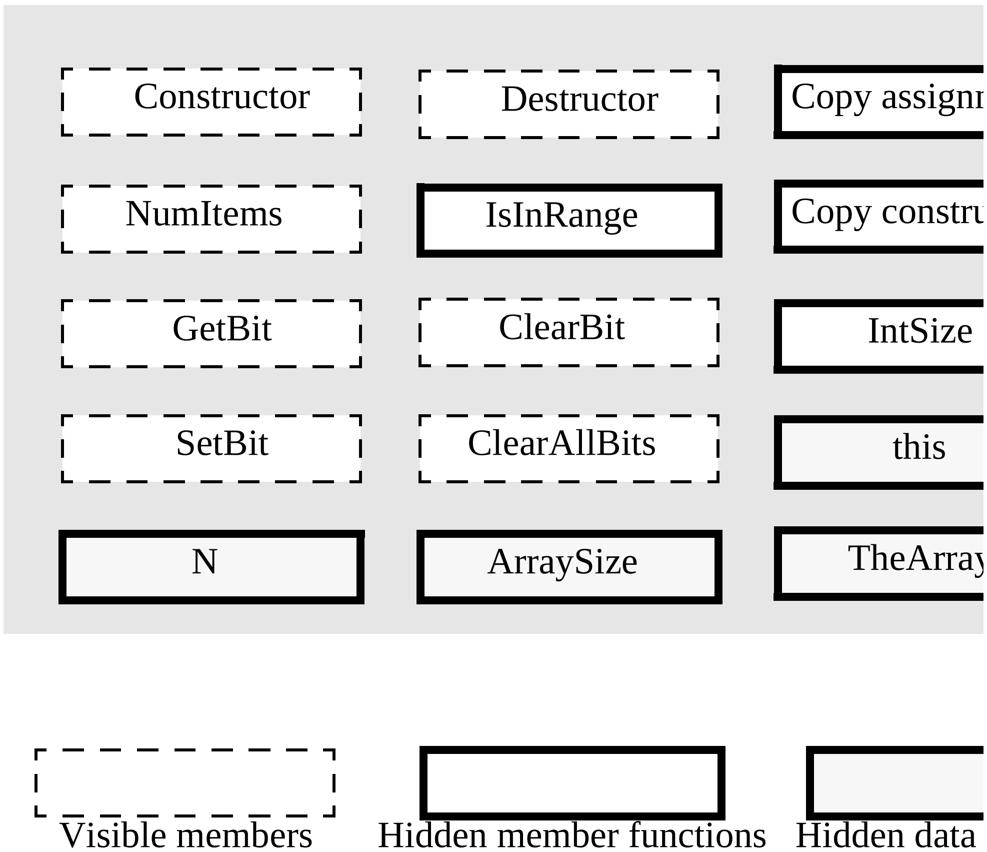
A simple test routine to show how `MemoryCell` objects are accessed


```
// MemoryCell interface
//  int Read( )         -->  Returns the stored value
//  void Write( int X ) -->  X is stored

class MemoryCell
{
  public:
    int Read( );
    void Write( int X );
  private:
    int StoredValue;
};



// Implementation of the MemoryCell class members

int
MemoryCell::Read( )
{
    return StoredValue;
}

void
MemoryCell::Write( int X )
{
    StoredValue = X;
}
```

A more typical `MemoryCell` declaration in which interface and implementation are separated

```
// BitArray class: support access to an array of bits
//
// CONSTRUCTION: with (a) no initializer or (b) an integer
//     that specifies the number of bits
// All copying of BitArray objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS**********************
// void ClearAllBits( )   --> Set all bits to zero
// void SetBit( int i )   --> Turn bit i on
// void ClearBit( int i ) --> Turn bit i off
// int GetBit( int i )    --> Return status of bit i
// int NumItems( )        --> Return capacity of bit array

#include <iostream.h>

class BitArray
{
  public:
    // Constructor
    BitArray( int Size = 320 );          // Basic constructor

    // Destructor
    ~BitArray( ) { delete [ ] TheArray; }

    // Member Functions
    void ClearAllBits( );
    void SetBit( int i );
    void ClearBit( int i );
    int  GetBit( int i ) const;
    int  NumItems( ) const { return N; }
  private:
        // 3 data members
    int *TheArray;                       // The bit array
    int N;                               // Number of bits
    int ArraySize;                       // Size of the array

    enum { IntSz = sizeof( int ) * 8 };
    int IsInRange( int i ) const;// Check range with error msg

        // Disable operator= and copy constructor
    const BitArray & operator=( const BitArray & Rhs );
    BitArray( const BitArray & Rhs );
};
```

Interface for `BitArray` class



`BitArray` members


```
BitArray A;              // Call with Size = 320
BitArray B( 50 );        // Call with Size = 50
BitArray C = 50;         // Same as above
BitArray D[ 50 ];        // Calls 50 constructors, with Size 320
BitArray *E = new BitArray; // Allocates BitArray of Size 320
E = new BitArray( 20 );// Allocates BitArray of size 20; leaks
BitArray F = "wrong";  // Does not match basic constructor
BitArray G( );           // This is wrong!
```

Construction examples

Chapter 3

Templates

Array position	0	1	2	3	4	5
Initial State:	8	5	9	2	6	3
After A[0..1] is sorted:	5	8	9	2	6	3
After A[0..2] is sorted:	5	8	9	2	6	3
After A[0..3] is sorted:	2	5	8	9	6	3
After A[0..4] is sorted:	2	5	6	8	9	3
After A[0..5] is sorted:	2	3	5	6	8	9

Basic action of insertion sort (shaded part is sorted)

Array position	0	1	2	3	4	5
Initial State:	8	5				
After A[0..1] is sorted:	5	8	9			
After A[0..2] is sorted:	5	8	9	2		
After A[0..3] is sorted:	2	5	8	9	6	
After A[0..4] is sorted:	2	5	6	8	9	3
After A[0..5] is sorted:	2	3	5	6	8	9

Closer look at action of insertion sort (dark shading indicates sorted area; light shading is where new element was placed)


```
// Typical template interface
template <class Etype>
class ClassName
{
  public:
    // Public members
  private:
    // Private members
};


// Typical member implementation
template <class Etype>
ReturnType
ClassName<Etype>::MemberName( Parameter List  ) /* const */
{
    // Member body
}
```

Typical layout for template interface and member functions

Chapter 4

Inheritance

```
class Derived : public Base
{
    // Any members that are not listed are inherited unchanged
    // except for constructor, destructor,
    // copy constructor, and operator=
  public:
    // Constructors, and destructors if defaults are not good
    // Base members whose definitions are to change in Derived
    // Additional public member functions
  private:
    // Additional data members (generally private)
    // Additional private member functions
    // Base members that should be disabled in Derived
};
```

General layout of public inheritance

Public inheritance situation	**Public**	**Protected**	**Private**
Base class member function accessing *M*	Yes	Yes	Yes
Derived class member function accessing *M*	Yes	Yes	No
`main`, accessing *B.M*	Yes	No	No
`main`, accessing *D.M*	Yes	No	No
Derived class member function accessing	Yes	No	No
B is an object of the base class; *D* is an object of the publicly derived class; *M* is a member of the base class.			

Access rules that depend on what *M*'s visibility is in the base class

Public inheritance situation	Public	Protected	Private
F accessing *B.MB*	Yes	Yes	Yes
F accessing *D.MD*	Yes	No	No
F accessing *D.MB*	Yes	Yes	Yes
B is an object of the base class; *D* is an object of the publicly derived class; *MB* is a member of the base class. *MD* is a member of the derived class. *F* is a friend of the base class (but not the derived class)			

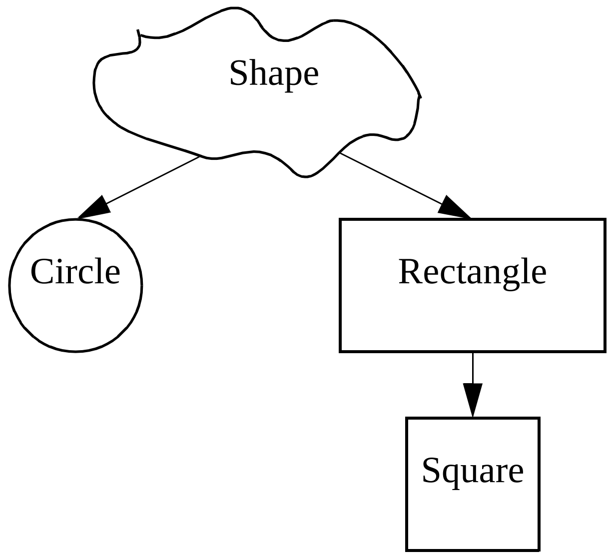
Friendship is not inherited

```
const VectorSize = 20;
Vector<int> V( VectorSize );
BoundedVector<int> BV( VectorSize, 2 * VectorSize - 1 );
    ...
BV[ VectorSize ] = V[ 0 ];
```

`Vector` and `BoundedVector` classes with calls to `operator[]` that are done automatically and correctly

```
Vector<int> *Vptr;
const int Size = 20;
cin >> Low;
if( Low )
    Vptr = new BoundedVector<int>( Low, Low + Size - 1 );
else
    Vptr = new Vector<int>( Size )

    ...
(*Vptr)[ Low ] = 0;       // What does this mean?
```

`Vector` and `BoundedVector` classes



The hierarchy of shapes used in an inheritance example

1. *Nonvirtual functions*: Overloading is resolved at compile time. To ensure consistency when pointers to objects are used, we generally use a nonvirtual function only when the function is invariant over the inheritance hierarchy (that is, when the function is never redefined). The exception to this rule is that constructors are always nonvirtual, as mentioned in Section 4.5.
2. *Virtual functions*: Overloading is resolved at run time. The base class provides a default implementation that may be overridden by the derived classes. Destructors should be virtual functions, as mentioned in Section 4.5.
3. *Pure virtual functions*: Overloading is resolved at run time. The base class provides no implementation. The absence of a default requires that the derived classes provide an implementation.

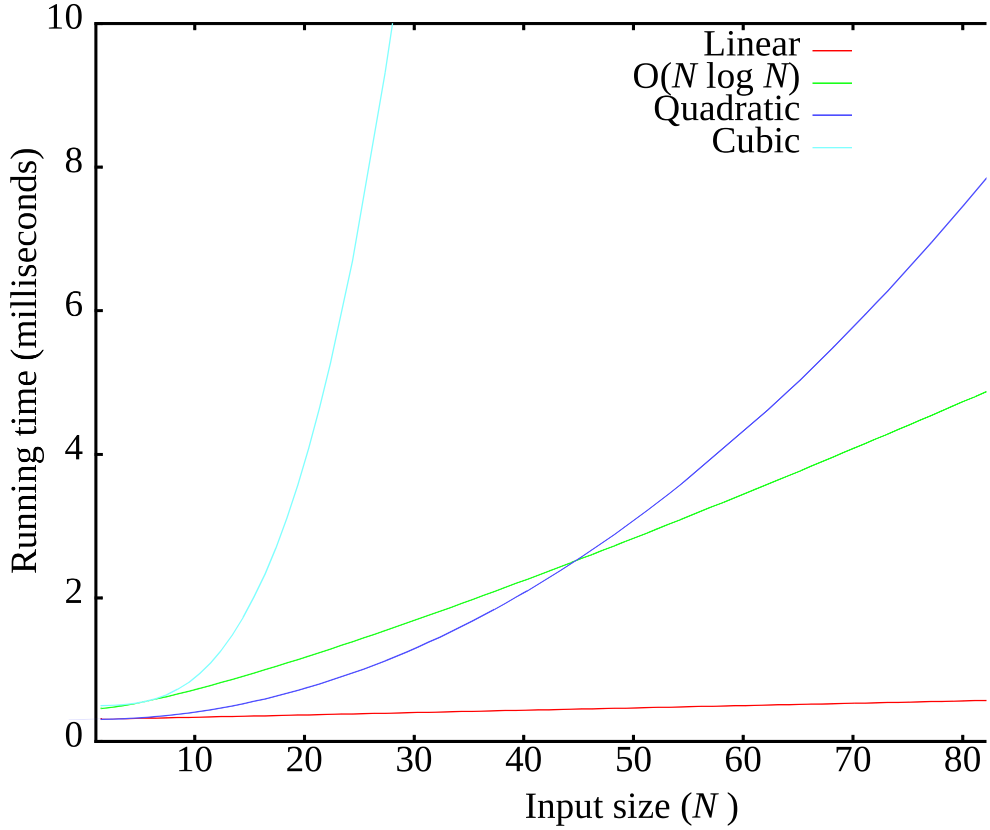
Summary of nonvirtual, virtual, and pure virtual functions

1. Provide a new constructor.
2. Examine each virtual function to decide if we are willing to accept its defaults; for each virtual function whose defaults we do not like, we must write a new definition.
3. Write a definition for each pure virtual function.
4. Write additional member functions if appropriate.

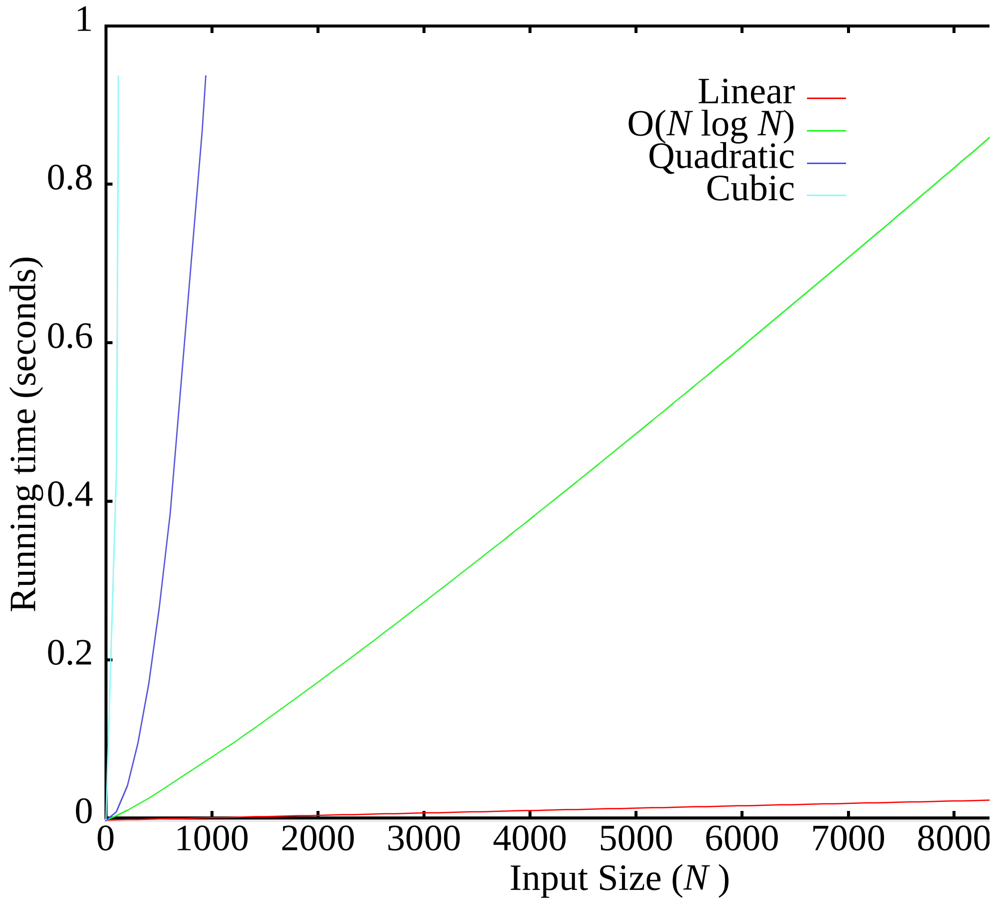
Programmer responsibilities for derived class

Chapter 5

Algorithm Analysis



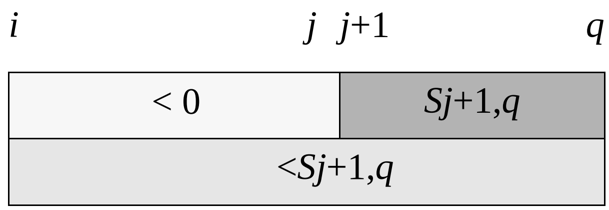
Running times for small inputs



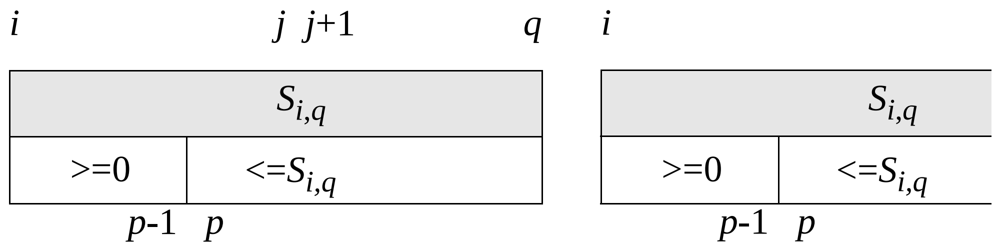
Running time for moderate inputs

Function	**Name**
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate



The subsequences used in Theorem 5.2



The subsequences used in Theorem 5.3. The sequence from p to q has sum at most that of the subsequence from i to q. On the left, the sequence from i to q is itself not the maximum (by Theorem 5.2). On the right, the sequence from i to q has already been seen.

DEFINITION: (Big-Oh) $T(\quad) = O(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) \le cF(\quad)$ when $N \ge N_0$.

DEFINITION: (Big-Omega) $T(\quad) = \Omega(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) \ge cF(\quad)$ when $N \ge N_0$.

DEFINITION: (Big-Theta) $T(\quad) = \Theta(\qquad)$ if and only if $T(\quad) = O(\qquad)$ and $T(\quad) = \Omega(\qquad)$.

DEFINITION: (Little-Oh) $T(\quad) = o(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) < cF(\quad)$ when $N \ge N_0$.

Mathematical expression	Relative rates of growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is = growth of $F(N)$
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$

Meanings of the various growth functions

N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.00103	0.00045	0.00066	0.00034
100	0.47015	0.01112	0.00486	0.00063
1,000	448.77	1.1233	0.05843	0.00333
10,000	NA	111.13	0.68631	0.03042
100,000	NA	NA	8.01130	0.29832

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

N	**CPU time** T **(milliseconds)**	T/N	T/N^2	$T/(N \log N)$
10,000	100	0.01000000	0.00000100	0.00075257
20,000	200	0.01000000	0.00000050	0.00069990
40,000	440	0.01100000	0.00000027	0.00071953
80,000	930	0.01162500	0.00000015	0.00071373
160,000	1960	0.01225000	0.00000008	0.00070860
320,000	4170	0.01303125	0.00000004	0.00071257
640,000	8770	0.01370313	0.00000002	0.00071046

Empirical running time for *N* binary searches in an *N*-item array

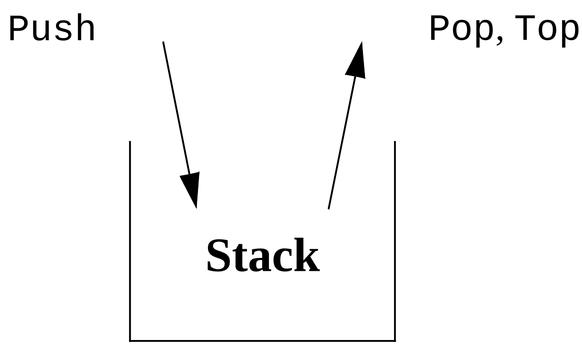
Chapter 6

Data Structures

```
#include <iostream.h>
#include "Stack.h"

// Simple test program for stacks

main( )
{
    Stack<int> S;

    for( int i = 0; i < 5; i++ )
        S.Push( i );

    cout << "Contents:";
    do
    {
        cout << ' ' << S.Top( );
        S.Pop( );
    } while( !S.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample stack program; output is

```
Contents: 4 3 2 1 0
```

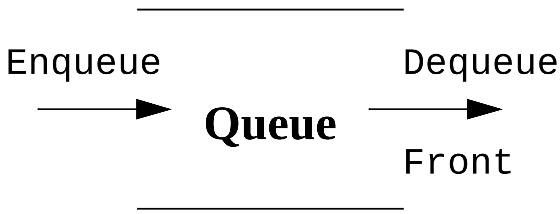


Stack model: input to a stack is by `Push`, output is by `Top`, deletion is by `Pop`

```
#include <iostream.h>
#include "Queue.h"

// Simple test program for queues

main( )
{
    Queue<int> Q;

    for( int i = 0; i < 5; i++ )
        Q.Enqueue( i );

    cout << "Contents:";
    do
    {
        cout << ' ' << Q.Front( );
        Q.Dequeue( );
    } while( !Q.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample queue program; output is

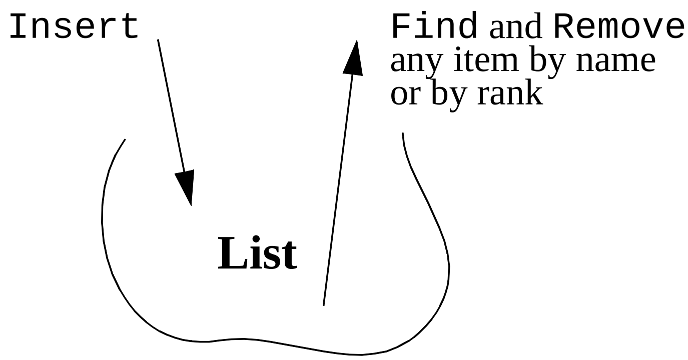
```
Contents:0 1 2 3 4
```

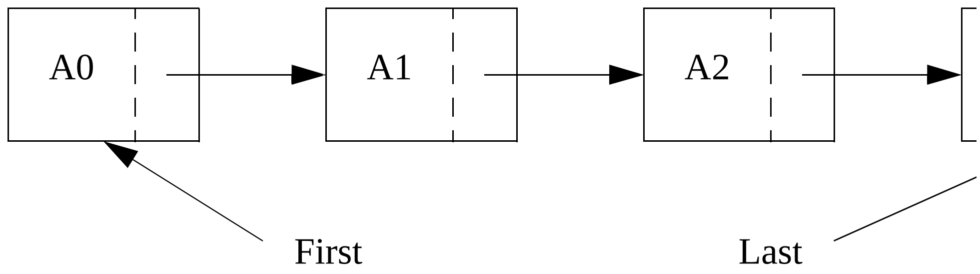
Queue model: input is by `Enqueue`, output is by `Front`, deletion is by `Dequeue`


```
#include <iostream.h>
#include "List.h"

// Simple test program for lists

main( )
{
    List<int> L;
    ListItr<int> P = L;

        // Repeatedly insert new items as first elements
    for( int i = 0; i < 5; i++ )
    {
        P.Insert( i );
        P.Zeroth( ); // Reset P to the start
    }

    cout << "Contents:";
    for( P.First( ); +P; ++P )
        cout << ' ' << P( );
    cout << "end\n";

    return 0;
}
```

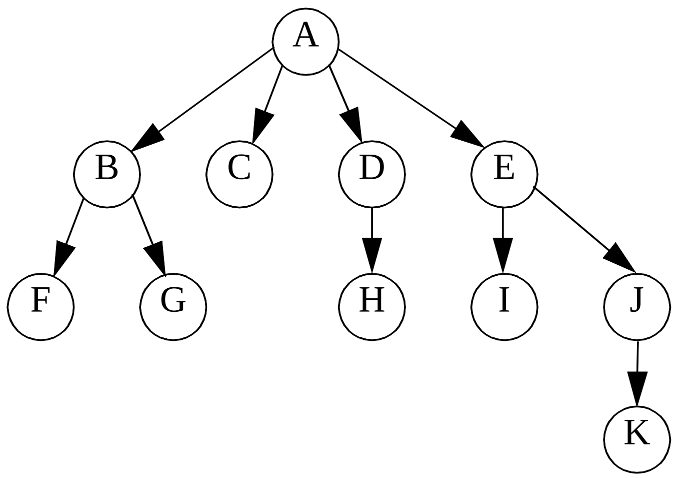
Sample list program; output is `Contents: 4 3 2 1 0 end`



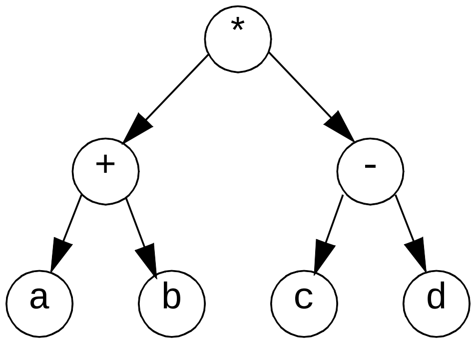
Link list model: inputs are arbitrary and ordered, any item may be output, and iteration is supported, but this data structure is not time-efficient



A simple linked list



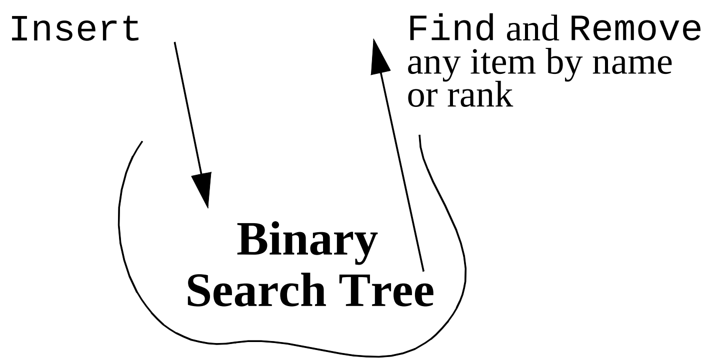
A tree



Expression tree for `(a+b)*(c-d)`

```
#include <iostream.h>
#include "Bst.h"

// Simple test program for binary search trees

main( )
{
    SearchTree<String> T;

    T.Insert( "Becky" );

        // Simple use of Find/WasFound
        // Appropriate if we need a copy
    String Result1 = T.Find( "Becky" );
    if( T.WasFound( ) )
        cout << "Found " << Result1 << ';';
    else
        cout << "Becky not found;";

        // More efficient use of Find/WasFound
        // Appropriate if we only need to examine
    const String & Result2 = T.Find( "Mark" );
    if( T.WasFound( ) )
        cout << " Found " << Result2 << ';';
    else
        cout << " Mark not found; ";

    cout << '\n';

    return 0;
}
```

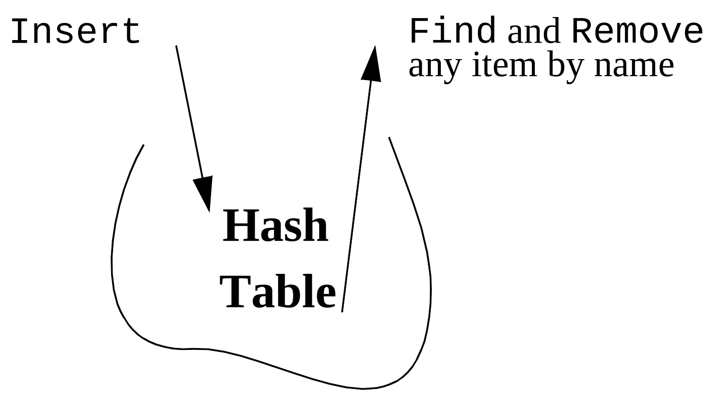
Sample search tree program;
output is `Found Becky; Mark not found;`



Binary search tree model; the binary search is extended to allow insertions and deletions


```
#include <iostream.h>
#include "Hash.h"

// A good hash function is given in Chapter 19
unsigned int Hash( const String & Element, int TableSize );

// Simple test program for hash tables

main( )
{
    HashTable<String> H;

    H.Insert( "Becky" );

    const String & Result2 = H.Find( "Mark" );
    if( H.WasFound( ) )
        cout << " Found " << Result2 << ';';
    else
        cout << " Mark not found; ";

    cout << '\n';

    return 0;
}
```

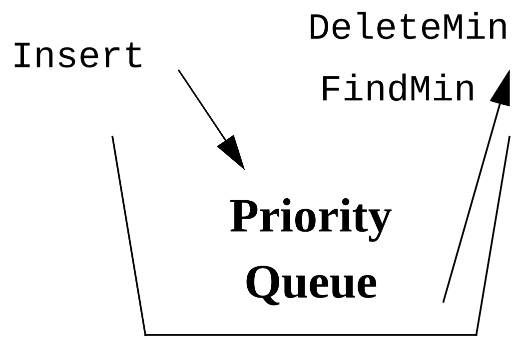
Sample hash table program;
output is `Found Becky; Mark not found;`



The hash table model: any named item can be accessed or deleted in essentially constant time

```
#include <iostream.h>
#include "BinaryHeap.h"

// Simple test program for priority queues

main( )
{
    BinaryHeap<int> PQ;

    PQ.Insert( 4 ); PQ.Insert( 2 ); PQ.Insert( 1 );
    PQ.Insert( 5 ); PQ.Insert( 0 );

    cout << "Contents:";
    do
    {
        cout << ' ' << PQ.FindMin( );
        PQ.DeleteMin( );
    } while( !PQ.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample program for priority queues;
output is `Contents: 0 1 2 3 4`



Priority queue model: only the minimum element is accessible

Data Structure	Access	Comments
Stack	Most recent only, `Pop`, $O(1)$	Very very fast
Queue	Least recent only, `Dequeue`, $O(1)$	Very very fast
Linked list	Any item	$O(N)$
Search Tree	Any item by name or rank, $O(\log N)$	Average case, can be made worst case
Hash Table	Any named item, $O(1)$	Almost certain
Priority Queue	`FindMin`, $O(1)$, `DeleteMin`, $O(\log N)$	`Insert` is $O(1)$ on average $O(\log N)$ worst case

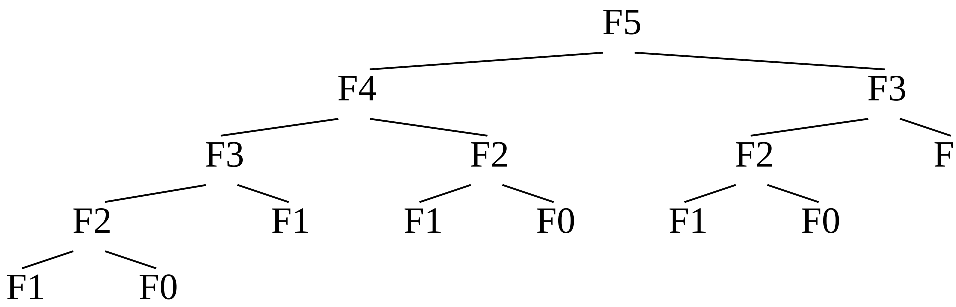
Summary of some data structures

Chapter 7

Recursion

TOP:	`S(2)`
	`S(3)`
	`S(4)`
	`main()`

Stack of activation records



Trace of the recursive calculation of the Fibonacci numbers

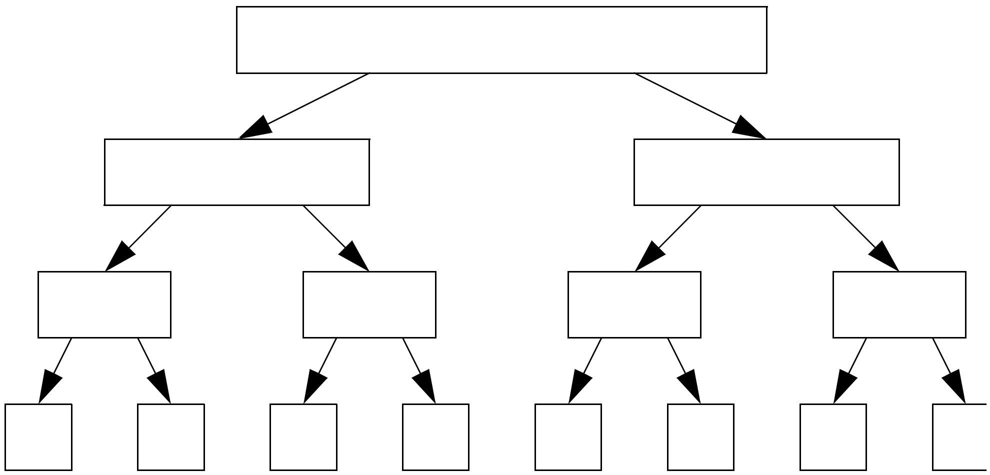
- *Divide*: Smaller problems are solved recursively (except, of course, base cases).
- *Conquer*: The solution to the original problem is then formed from the solutions to the subproblems.

Divide-and-conquer algorithms

First Half				Second Half				
4	-3	5	-2	-1	2	6	-2	Values
4*	0	3	-2	-1	1	7*	5	Running Sums

Running Sum from the Center (*denotes maximum for each half)

Dividing the maximum contiguous subsequence problem into halves



Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm

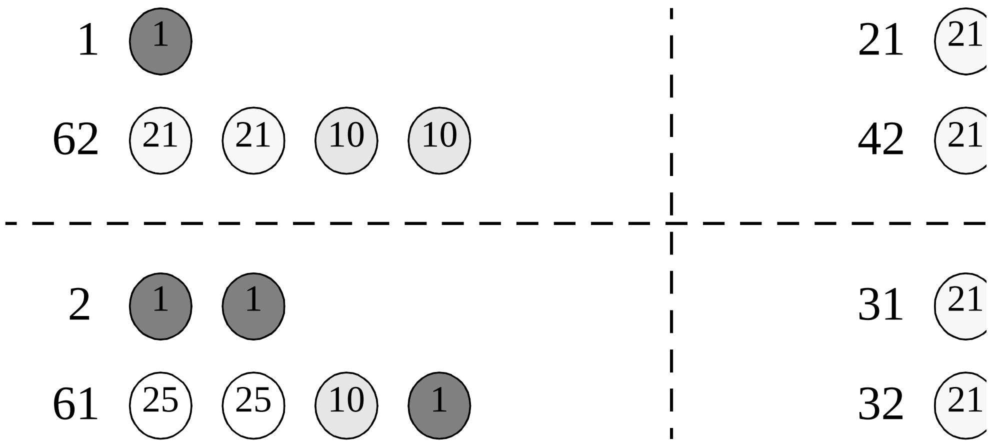
Assuming N is a power of 2, the solution to the equation $T(N) = 2T(N/2) + N$, with initial condition $T(1) = 1$ is $T(N) = N \log N + N$.

Basic divide-and-conquer running time theorem

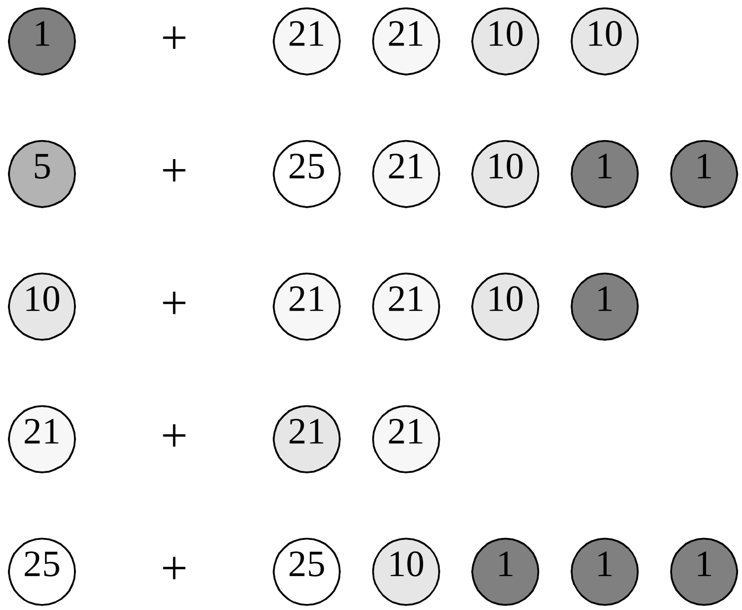
The solution to the equation $T(\quad) = AT(\qquad) + O(\quad)$, where $A \ge 1$ and $B > 1$, is

$$T(\quad) = \begin{cases} O(\qquad) & \text{if } A > B^k \\ O(\qquad) & \text{if } A = B^k \\ O(\quad) & \text{if } A < B^k \end{cases}$$

General divide-and-conquer running time theorem



Some of the subproblems that are solved recursively in Figure 7.15



Alternative recursive algorithm for coin-changing problem

Chapter 8

Sorting Algorithms

- Words in a dictionary are sorted (and case distinctions are ignored).
- Files in a directory are often listed in sorted order.
- The index of a book is sorted (and case distinctions are ignored).
- The card catalog in a library is sorted by both author and title.
- A listing of course offerings at a university is sorted, first by department and then by course number.
- Many banks provide statements that list checks in increasing order (by check number).
- In a newspaper, the calendar of events in a schedule is generally sorted by date.
- Musical compact disks in a record store are generally sorted by recording artist.
- In the programs that are printed for graduation ceremonies, departments are listed in sorted order, and then students in those departments are listed in sorted order.

Examples of sorting

<table>
<tr><th>Operators</th><th>Definition</th></tr>
<tr><td><code>operator> (A, B)</code></td><td><code>return B < A;</code></td></tr>
<tr><td><code>operator>=(A, B)</code></td><td><code>return !(A < B);</code></td></tr>
<tr><td><code>operator<=(A, B)</code></td><td><code>return !(B < A);</code></td></tr>
<tr><td><code>operator!=(A, B)</code></td><td><code>return A < B || B < A;</code></td></tr>
<tr><td><code>operator==(A, B)</code></td><td><code>return !(A < B || B < A);</code></td></tr>
</table>

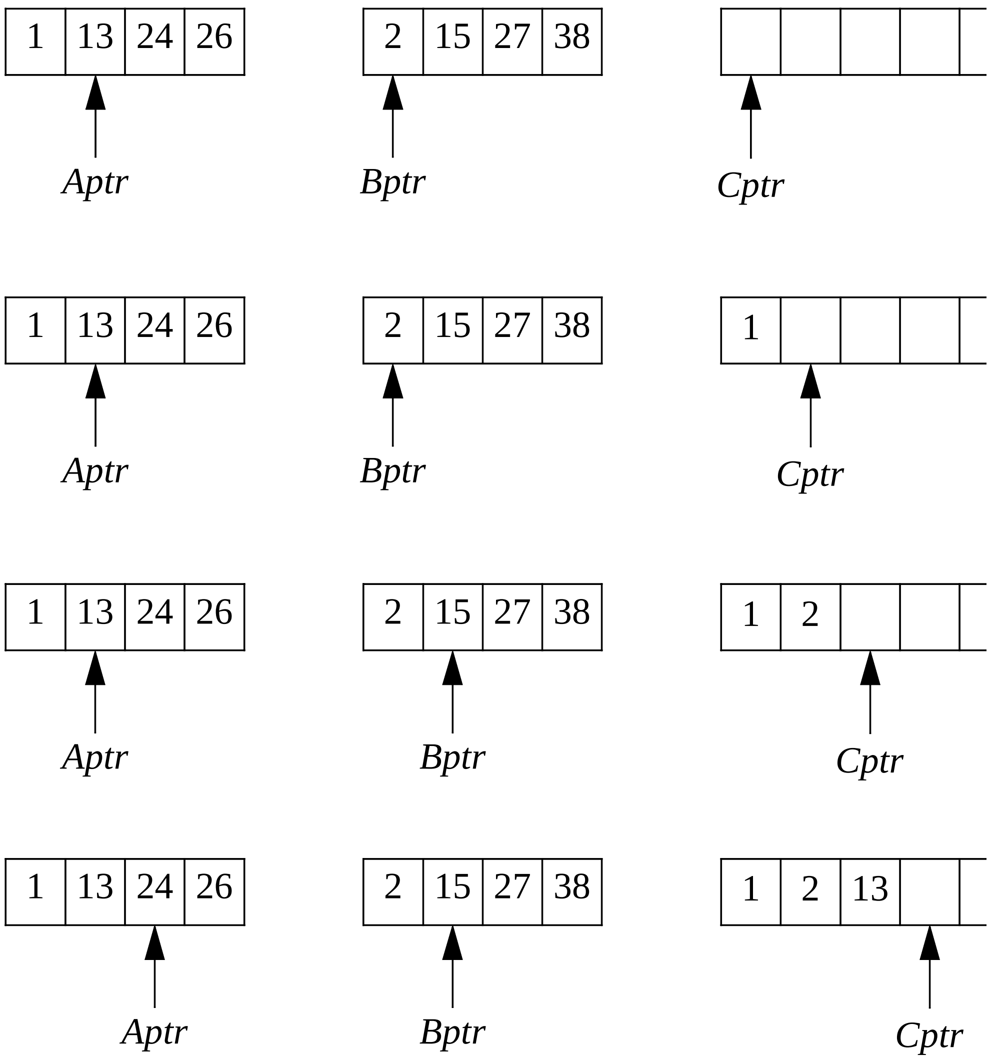
Deriving the relational and equality operators from `operator<`

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

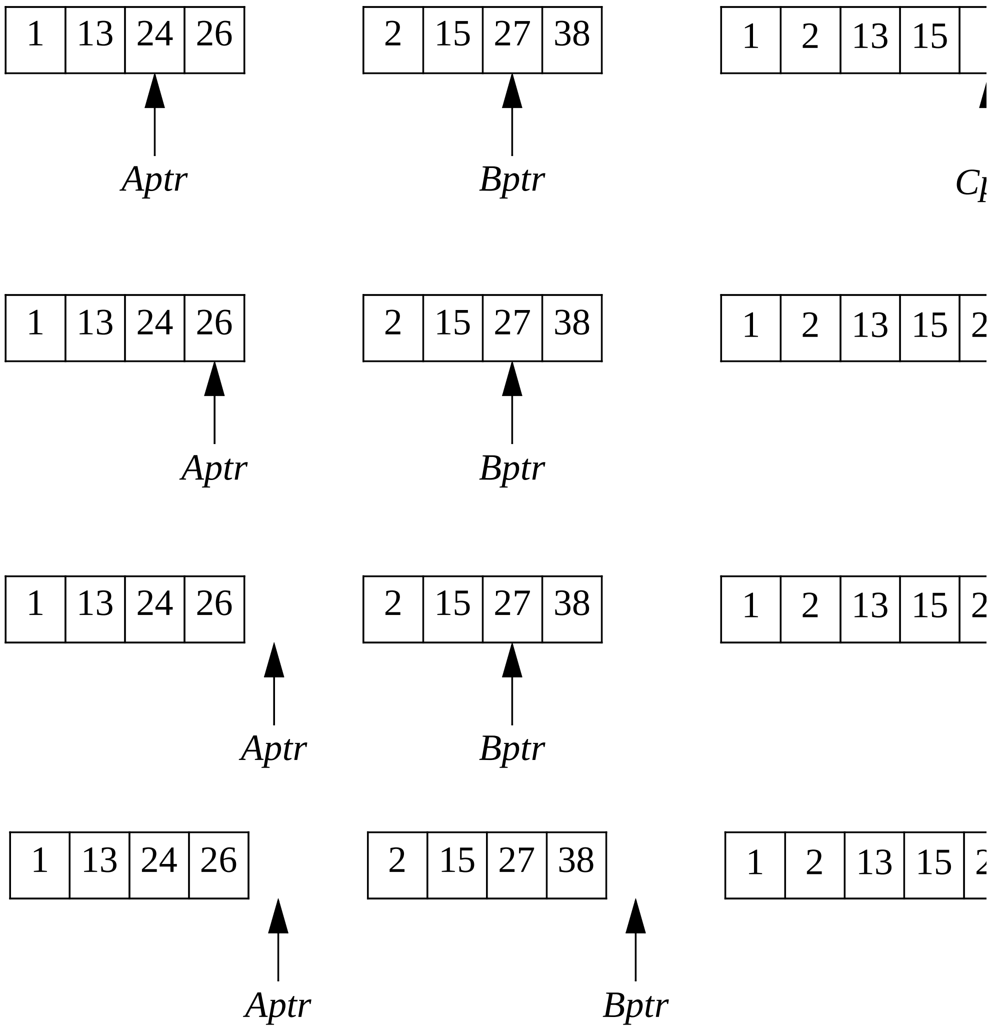
Shellsort after each pass, if increment sequence is {1, 3, 5}

N	Insertion sort	Shellsort		
		Shell's	Odd gaps only	Dividing by 2.2
1,000	122	11	11	9
2,000	483	26	21	23
4,000	1,936	61	59	54
8,000	7,950	153	141	114
16,000	32,560	358	322	269
32,000	131,911	869	752	575
64,000	520,000	2,091	1,705	1,249

Running time (milliseconds) of the insertion sort and Shellsort with various increment sequences



Linear-time merging of sorted arrays (first four steps)

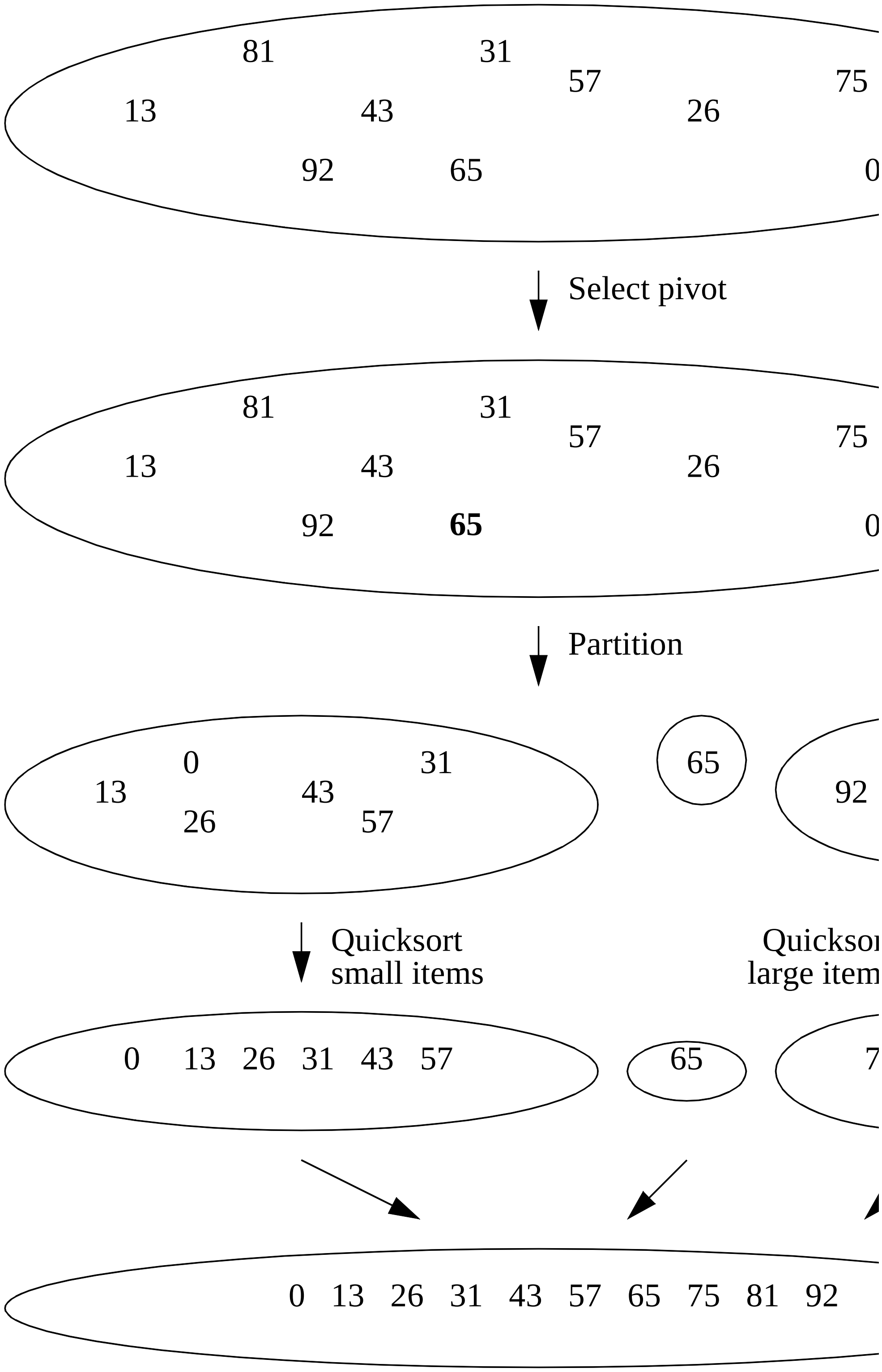


Linear-time merging of sorted arrays (last four steps)

The basic algorithm *Quicksort*(S) consists of the following four steps:

1. If the number of elements in S is 0 or 1, then return.
2. Pick *any* element v in S. This is called the *pivot*.
3. *Partition* $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $L = \{\qquad\qquad | \qquad \}$ and $R = \{x \in S - \{v\} | x \geq v\}$.
4. Return the result of *Quicksort*(L) followed by v followed by *Quicksort*(R).

Basic quicksort algorithm



The steps of quicksort

Because recursion allows us to take the giant leap of faith, the correctness of the algorithm is guaranteed as follows:

- The group of small elements is sorted, by virtue of the recursion.
- The largest element in the group of small elements is not larger than the pivot, by virtue of the partition.
- The pivot is not larger than the smallest element in the group of large elements, by virtue of the partition.
- The group of large elements is sorted, by virtue of the recursion.

Correctness of quicksort

8	1	4	9	0	3	5	2	7	6

Partitioning algorithm: pivot element 6 is placed at the end

8	1	4	9	0	3	5	2	7	6

Partitioning algorithm: `i` stops at large element 8; `j` stops at small element 2

2	1	4	9	0	3	5	8	7	6

Partitioning algorithm: out-of-order elements 8 and 2 are swapped

2	1	4	9	0	3	5	8	7	6

Partitioning algorithm: `i` stops at large element 9; `j` stops at small element 5

2	1	4	5	0	3	9	8	7	6

Partitioning algorithm: out-of-order elements 9 and 5 are swapped

2	1	4	5	0	3	9	8	7	6

Partitioning algorithm: `i` stops at large element 9; `j` stops at small element 3

2	1	4	5	0	3	6	8	7	9

Partitioning algorithm: swap pivot and element in position `i`

8	1	4	9	6	3	5	2	7	0

Original array

0	1	4	9	6	3	5	2	7	8

Result of sorting three elements (first, middle, and last)

0	1	4	9	7	3	5	2	6	8

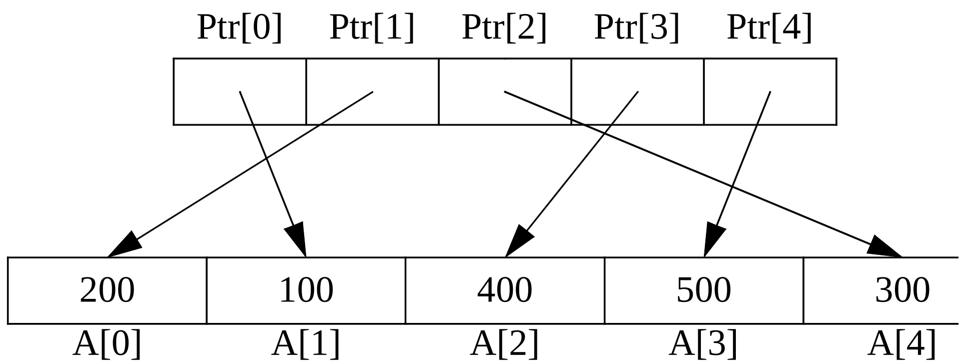
Result of swapping the pivot with next to last element

- We should not swap the pivot with the element in the last position. Instead, we should swap it with the element in the next to last position.
- We can start `i` at `Low+1` and `j` at `High-2`.
- We are guaranteed that, whenever `i` searches for a large element, it will stop because in the worst case it will encounter the pivot (and we stop on equality).
- We are guaranteed that, whenever `j` searches for a small element, it will stop because in the worst case it will encounter the first element (and we stop on equality).

Median-of-three partitioning optimizations

1. If the number of elements in S is 1, then presumably k is also 1, and we can return the single element in S.
2. Pick any element v in S. This is the pivot.
3. *Partition* $S - \{v\}$ into L and R, exactly as was done for quicksort.
4. If k is less than or equal to the number of elements in L, then the item we are searching for must be in L. Call *Quickselect(L, k)* recursively. Otherwise, if k is exactly equal to one more than the number of items in L, then the pivot is the kth smallest element, and we can return it as the answer. Otherwise, the *kth* smallest element lies in R, and it is the $(k - |L| - 1)$th smallest element in R. Again, we can make a recursive call and return the result.

Quickselect algorithm



Using an array of pointers to sort

Loc[0]	Loc[1]	Loc[2]	Loc[3]	Loc[4]
1	0	4	2	3

A[0]	A[1]	A[2]	A[3]	A[4]
200	100	400	500	300

Data structure used for in-place rearrangement

Chapter 9

Randomization

Winning Tickets	0	1	2	3	4	5
Frequency	0.135	0.271	0.271	0.180	0.090	0.036

Distribution of lottery winners if expected number of winners is 2

An important nonuniform distribution that occurs in simulations is the *Poisson distribution*. Occurrences that happen under the following circumstances satisfy the Poisson distribution:

- The probability of one occurrence in a small region is proportional to the size of the region.
- The probability of two occurrences in a small region is proportional to the square of the size of the region and is usually small enough to be ignored.
- The event of getting k occurrences in one region and the event of getting j occurrences in another region disjoint from the first region are independent. (Technically this statement means that you can get the probability of both events simultaneously occurring by multiplying the probability of individual events.)
- The mean number of occurrences in a region of some size is known.

Then if the mean number of occurrences is the constant a, then the probability of exactly k occurrences is $a^k e^{-a} / k!$.

Poisson distribution

Chapter 10

Fun and Games

	0	**1**	**2**	**3**
0	t	h	i	s
1	w	a	t	s
2	o	a	h	g
3	f	g	d	t

Sample word search grid

```
for each word W in the word list
    for each row R
        for each column C
            for each direction D
                check if W exists at row R, column C
                in direction D
```

Brute-force algorithm for word search puzzle

```
for each row R
    for each column C
        for each direction D
            for each word length L
                check if L chars starting at row R column C
                            in direction D form a word
```

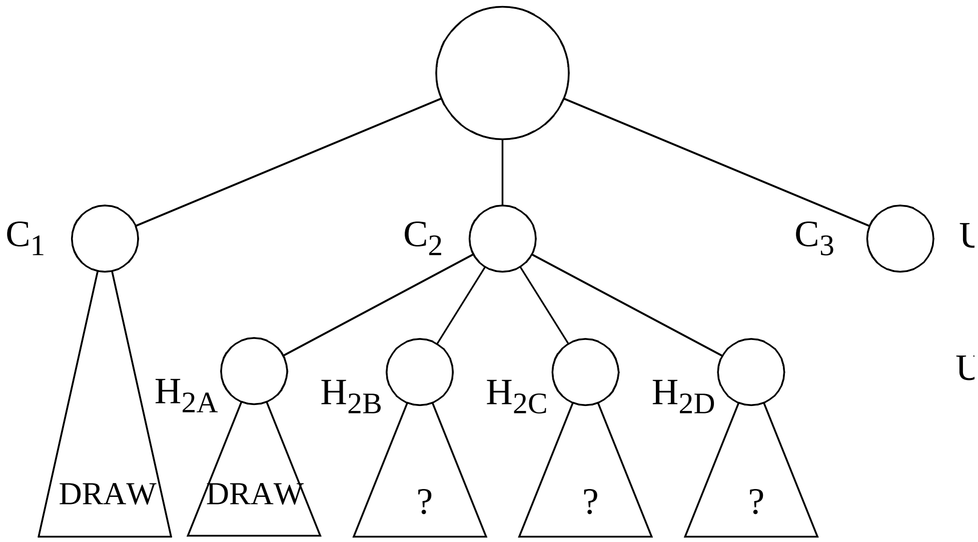
Alternate algorithm for word search puzzle

```
for each row R
    for each column C
        for each direction D
            for each word length L
                check if L chars starting at row R column
                            C in direction D form a word
                if they do not form a prefix,
                    break;   // the innermost loop
```

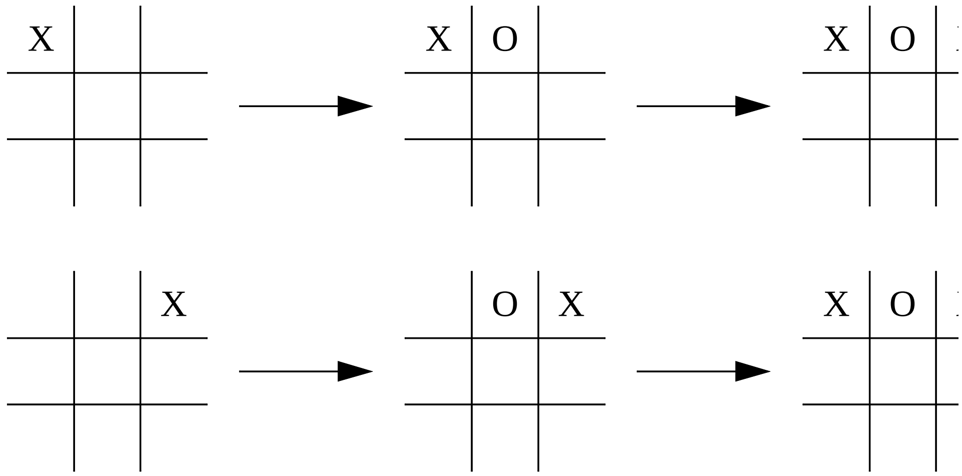
Improved algorithm for word search puzzle; incorporates a prefix test

1. If the position is *terminal* (that is, can immediately be evaluated), return its value.
2. Otherwise, if it is the computer's turn to move, return the maximum value of all positions reachable by making one move. The reachable values are calculated recursively.
3. Otherwise, it is the human's turn to move. Return the minimum value of all positions reachable by making one move. The reachable values are calculated recursively.

Basic minimax algorithm



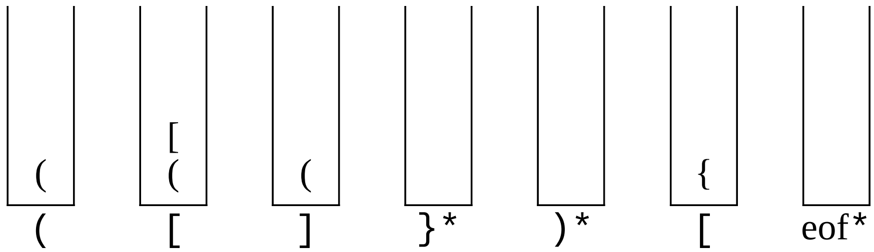
Alpha-beta pruning: After H_{2A} is evaluated, C_2, which is the minimum of the H_2's, is at best a draw. Consequently, it cannot be an improvement over C_1. We therefore do not need to evaluate H_{2B}, H_{2C}, and H_{2D}, and can proceed directly to C_3



Two searches that arrive at identical positions

Chapter 11

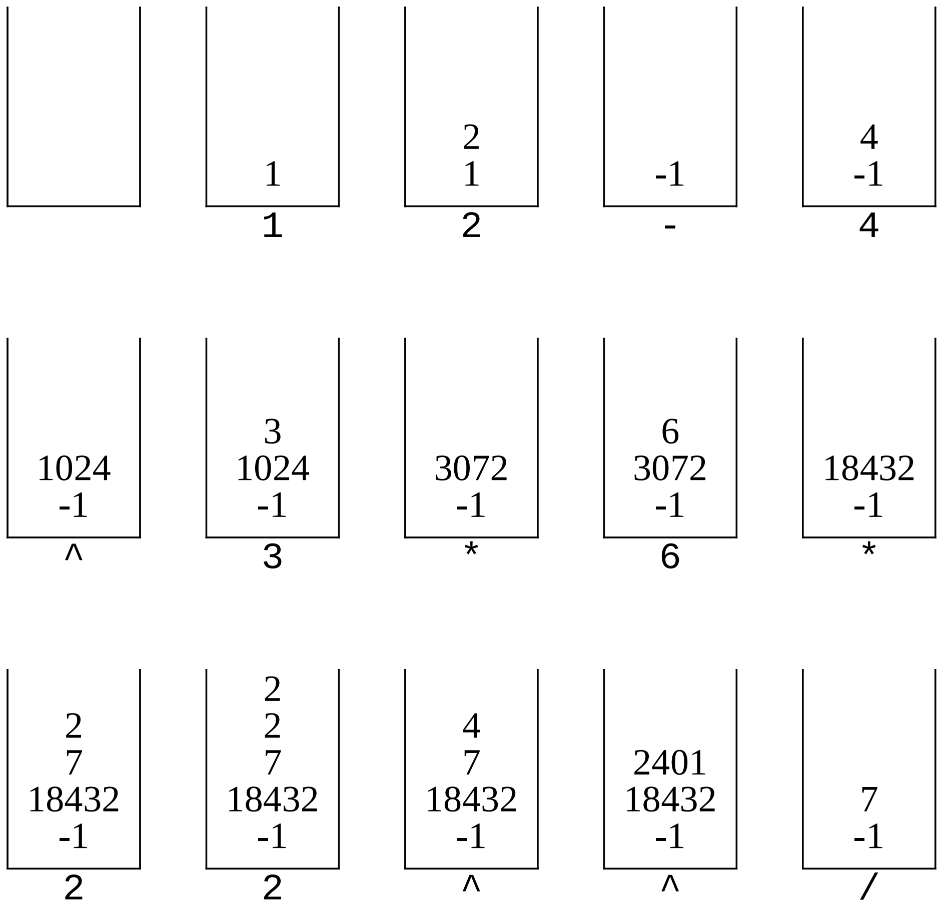
Stacks and Compilers




```
Errors (indicated by *):
    } when expecting )
    ) with no matching opening symbc
    [ unmatched at end of input
```

Stack operations in balanced symbol algorithm

Postfix Expression: `1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^`



Steps in evaluation of a postfix expression

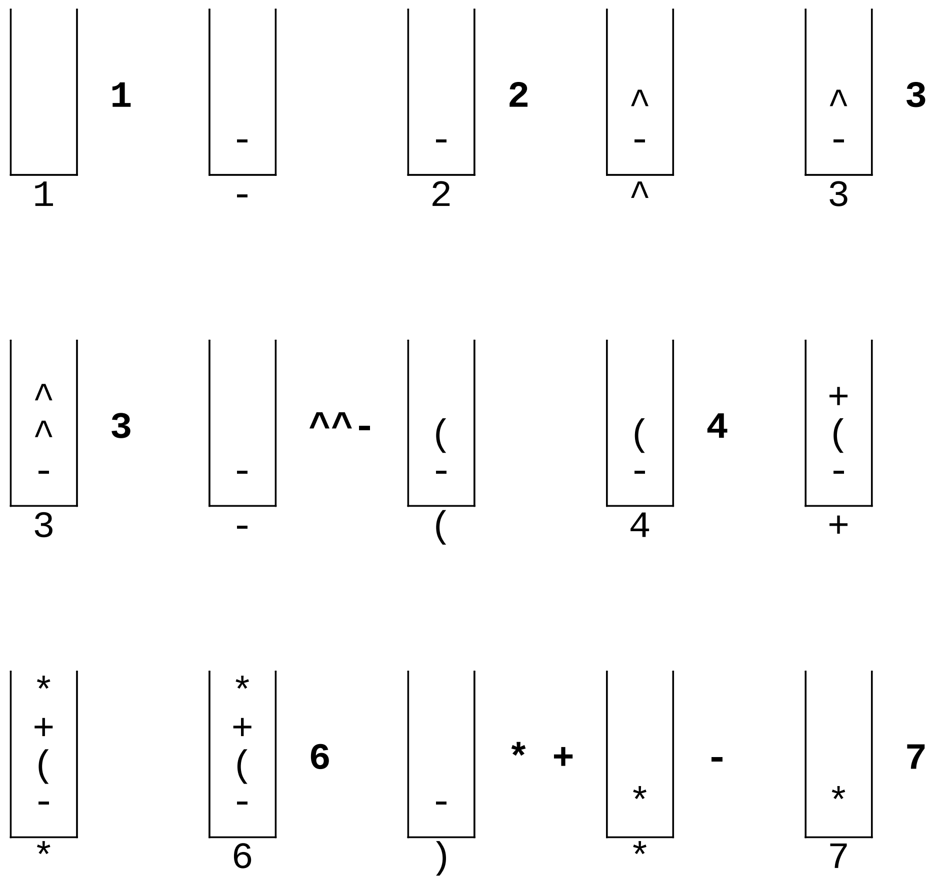
Infix expression	Postfix expression	Associativity
2 + 3 + 4	2 3 + 4 +	Left associative: Input + is lower than stack +
2 ^ 3 ^ 4	2 3 4 ^ ^	Right associative: Input ^ is higher than stack ^

Associativity rules

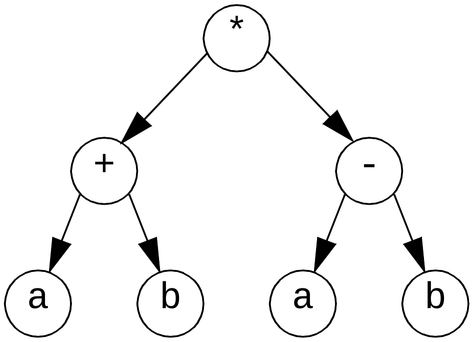
- *Operands*: Immediately output.
- *Close parenthesis*: Pop stack symbols until an open parenthesis is seen.
- *Operator*: Pop all stack symbols until we see a symbol of lower precedence or a right associative symbol of equal precedence. Then push the operator.
- *End of input*: Pop all remaining stack symbols.

Various cases in operator precedence parsing

Infix: `1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7`



Infix to postfix conversion



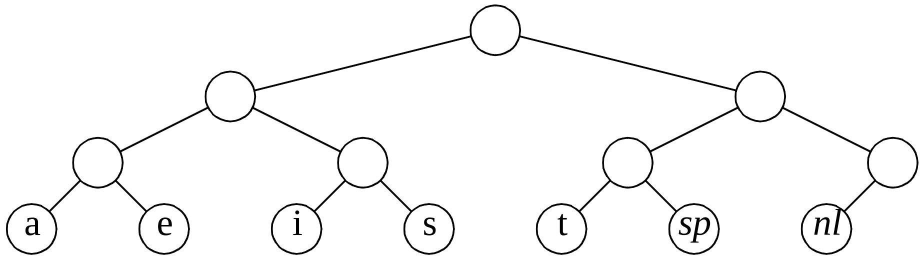
Expression tree for `(a+b)*(c-d)`

Chapter 12

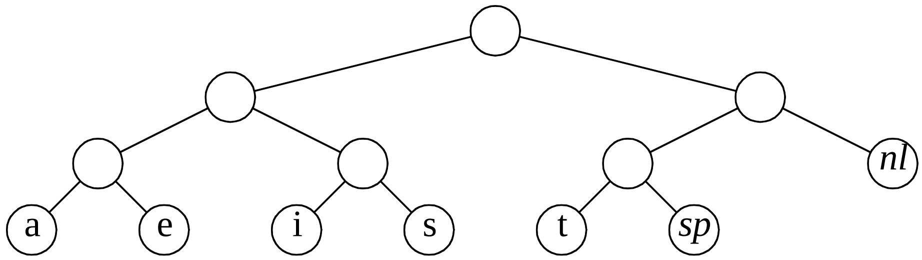
Utilities

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
Total			**174**

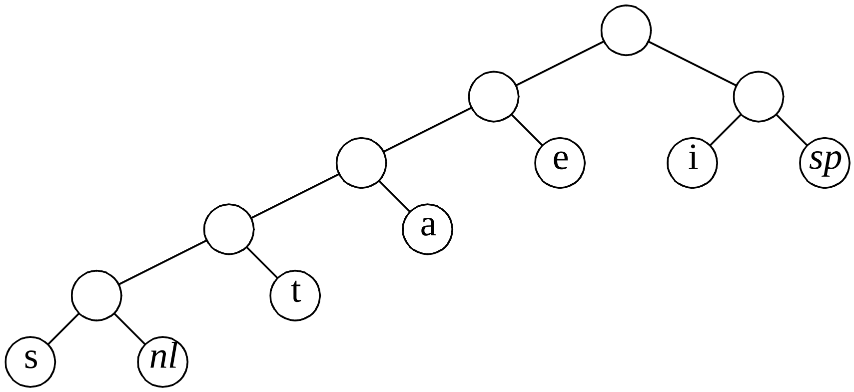
A standard coding scheme



Representation of the original code by a tree



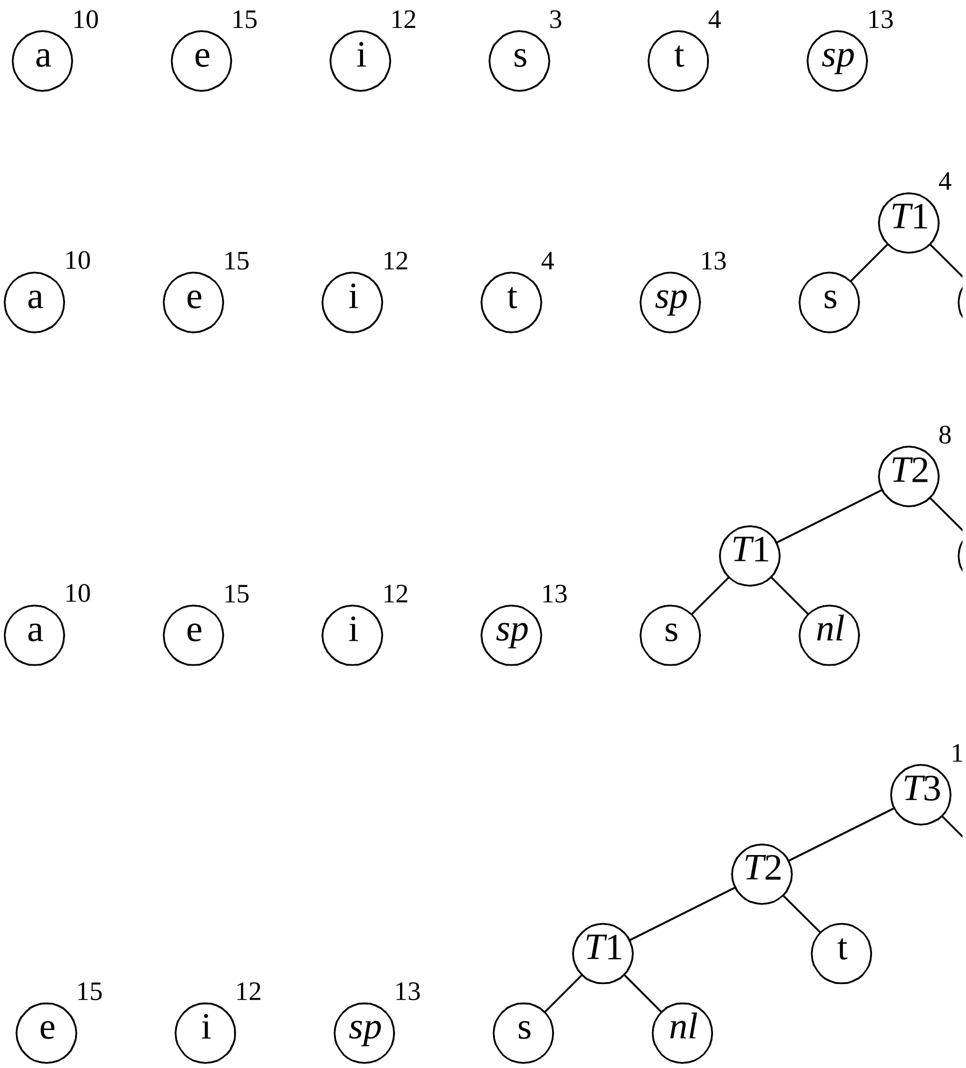
A slightly better tree



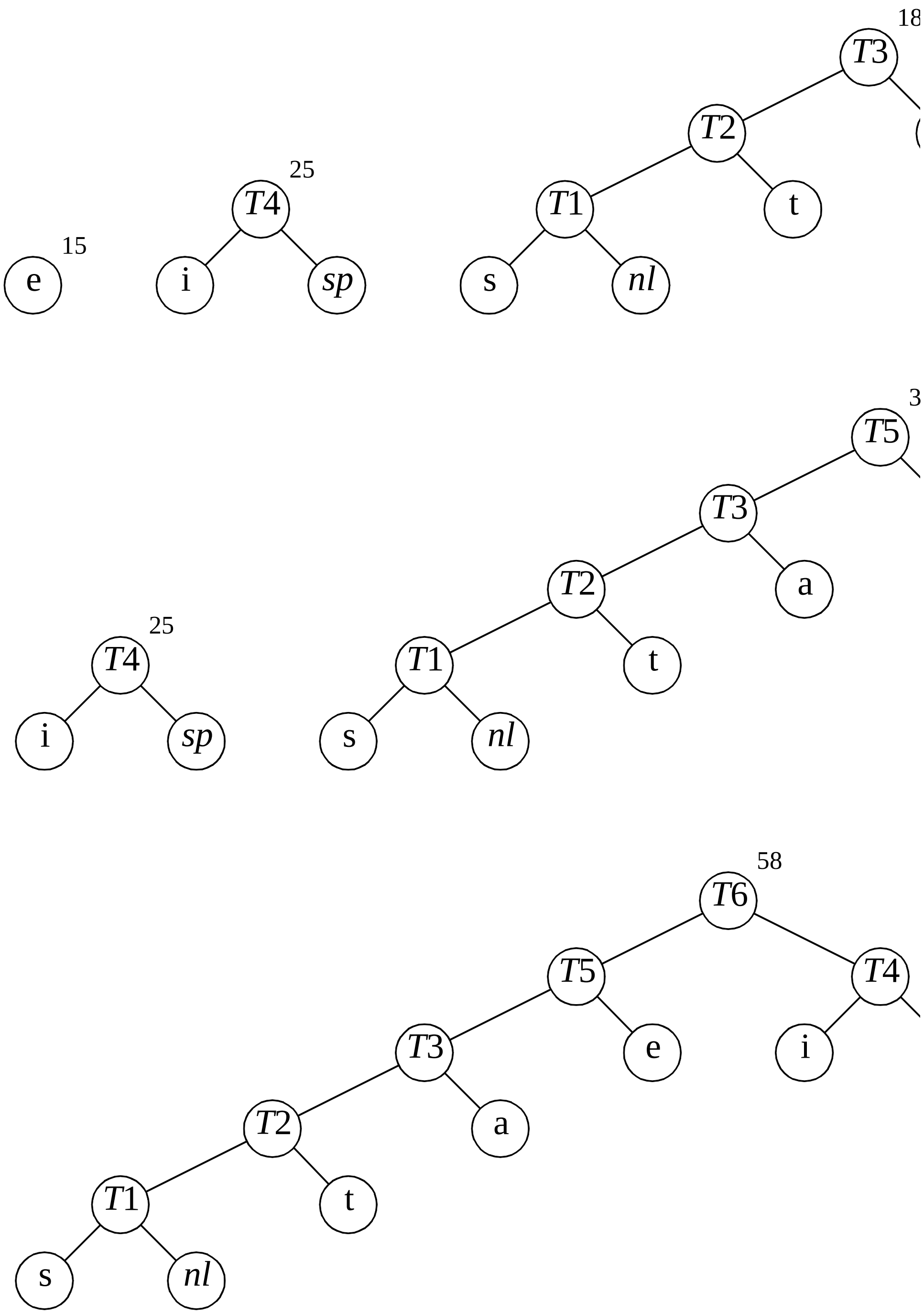
Optimal prefix code tree

Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			**146**

Optimal prefix code



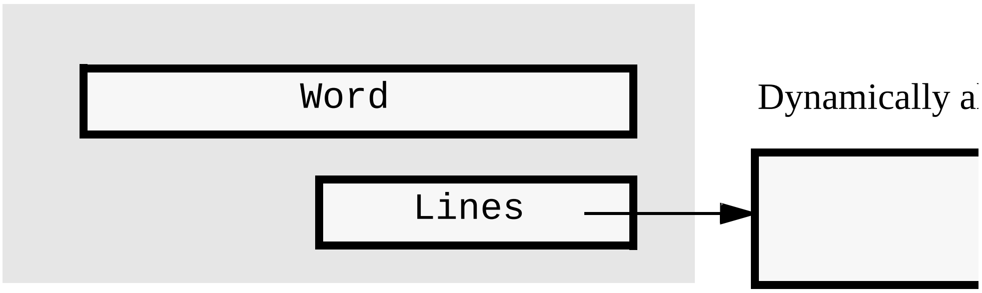
Huffman’s algorithm after each of first three merges



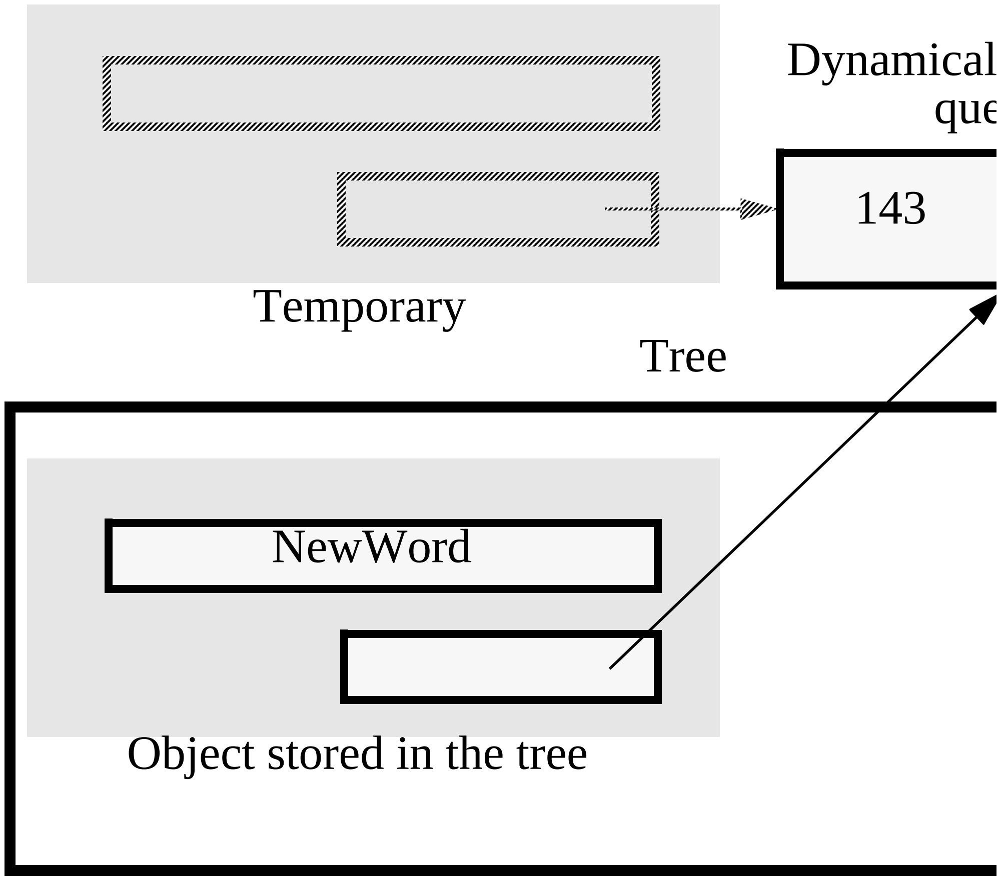
Huffman’s algorithm after each of last three merges

	Character	Weight	Parent	Child Type
0	a	10	9	1
1	e	15	11	1
2	i	12	10	0
3	s	3	7	0
4	t	4	8	1
5	sp	13	10	1
6	nl	1	7	1
7	T1	4	8	0
8	T2	8	9	0
9	T3	18	11	0
10	T4	25	12	1
11	T5	33	12	0
12	T6	58	0	

Encoding table (numbers on left are array indices)



`IdNode` data members: `Word` is a `String`; `Lines` is a pointer to a `Queue`

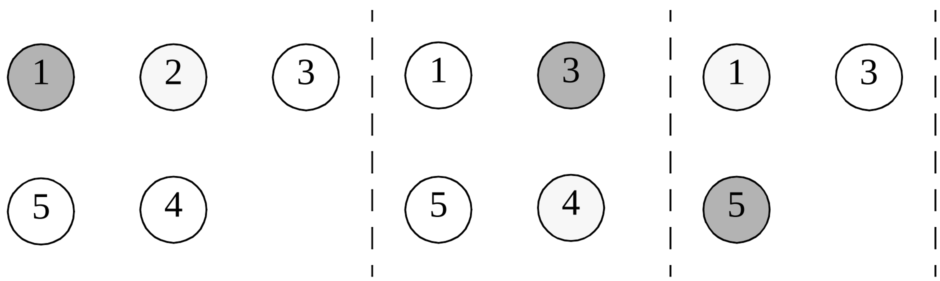


The object in the tree is a copy of the temporary; after the insertion is complete, the destructor is called for the temporary

Chapter 13

Simulation

1. At the start, the potato is at player 1; after one pass it is at player 2.
2. Player 2 is eliminated, player 3 picks up the potato, and after one pass it is at player 4.
3. Player 4 is eliminated, player 5 picks up the potato and passes it to player 1.
4. Player 1 is eliminated, player 3 picks up the potato, and passes it to player 5.
5. Player 5 is eliminated, so player 3 wins.



(a) (b) (c)

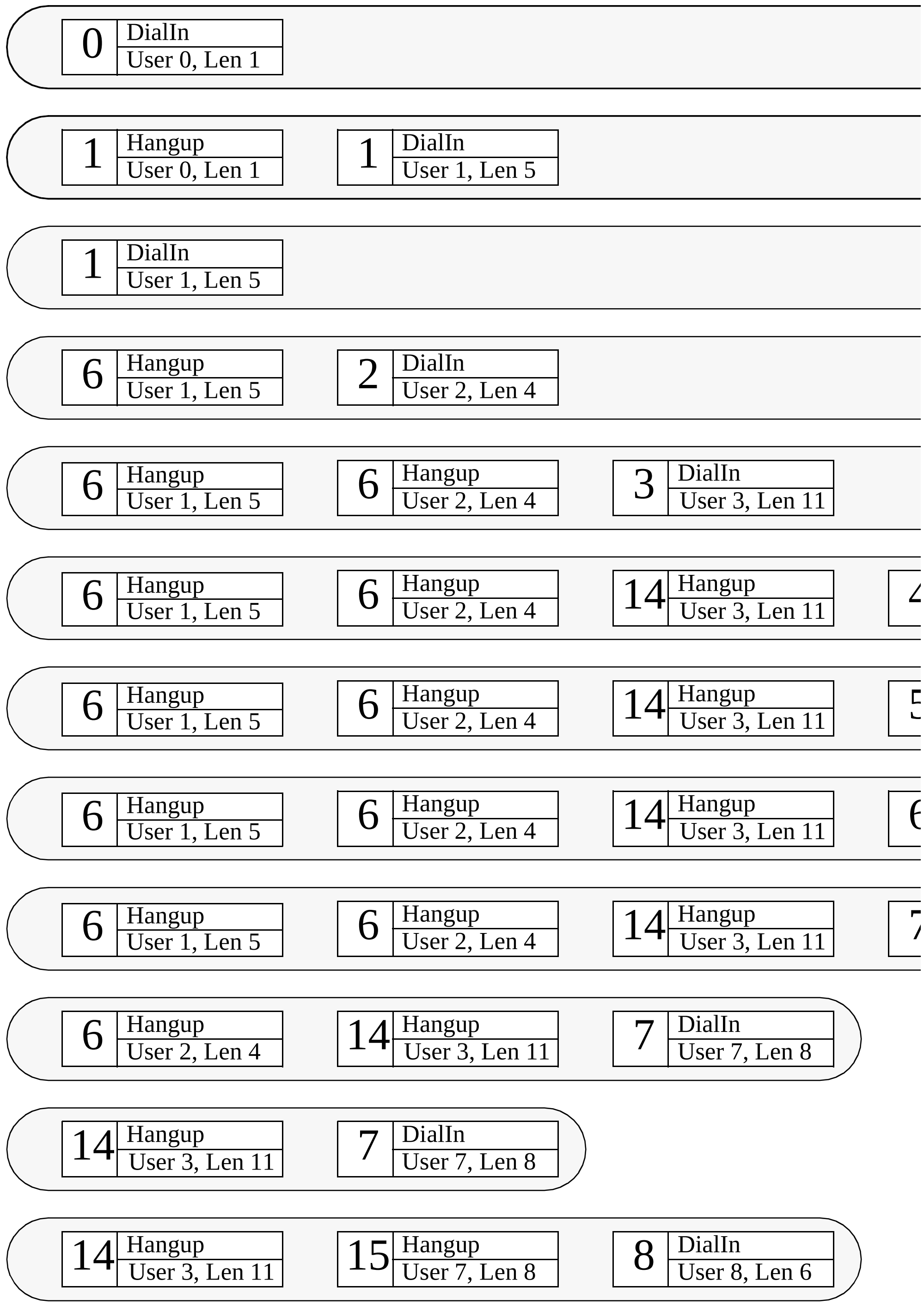
The Josephus problem


```
User 0 dials in at time 0 and connects for 1 minutes
User 0 hangs up at time 1
User 1 dials in at time 1 and connects for 5 minutes
User 2 dials in at time 2 and connects for 4 minutes
User 3 dials in at time 3 and connects for 11 minutes
User 4 dials in at time 4 but gets busy signal
User 5 dials in at time 5 but gets busy signal
User 6 dials in at time 6 but gets busy signal
User 1 hangs up at time 6
User 2 hangs up at time 6
User 7 dials in at time 7 and connects for 8 minutes
User 8 dials in at time 8 and connects for 6 minutes
User 9 dials in at time 9 but gets busy signal
User 10 dials in at time 10 but gets busy signal
User 11 dials in at time 11 but gets busy signal
User 12 dials in at time 12 but gets busy signal
User 13 dials in at time 13 but gets busy signal
User 3 hangs up at time 14
User 14 dials in at time 14 and connects for 6 minutes
User 8 hangs up at time 14
User 15 dials in at time 15 and connects for 3 minutes
User 7 hangs up at time 15
User 16 dials in at time 16 and connects for 5 minutes
User 17 dials in at time 17 but gets busy signal
User 15 hangs up at time 18
User 18 dials in at time 18 and connects for 7 minutes
User 19 dials in at time 19 but gets busy signal
```

Sample output for the modem bank simulation: 3 modems; a dial in is attempted every minute; average connect time is 5 minutes; simulation is run for 19 minutes

1. The first `DialIn` request is inserted
2. After `DialIn` is removed, the request is connected resulting in a `Hangup` and a replacement `DialIn` request
3. A `Hangup` request is processed
4. A `DialIn` request is processed resulting in a connect. Thus both a `Hangup` and `DialIn` event are added (three times)
5. A `DialIn` request fails; a replacement `DialIn` is generated (three times)
6. A `Hangup` request is processed (twice)
7. A `DialIn` request succeeds, `Hangup` and `DialIn` are added.

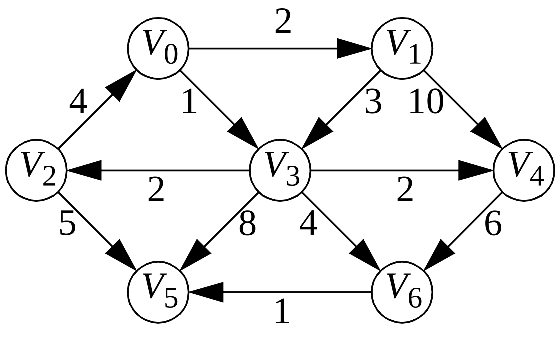
Steps in the simulation



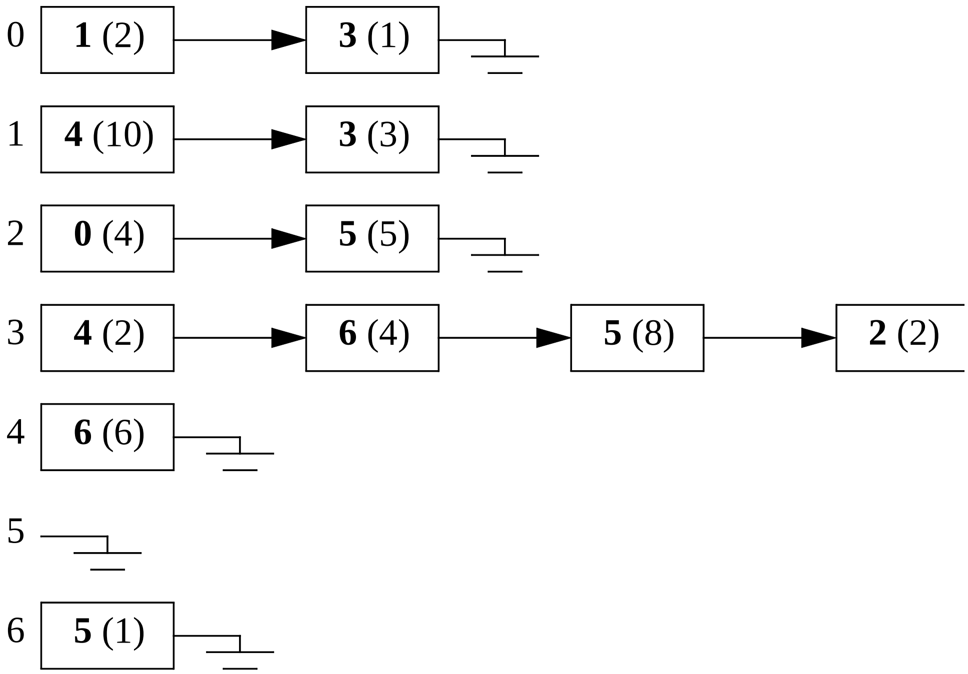
Priority queue for modem bank after each step

Chapter 14

Graphs and Paths



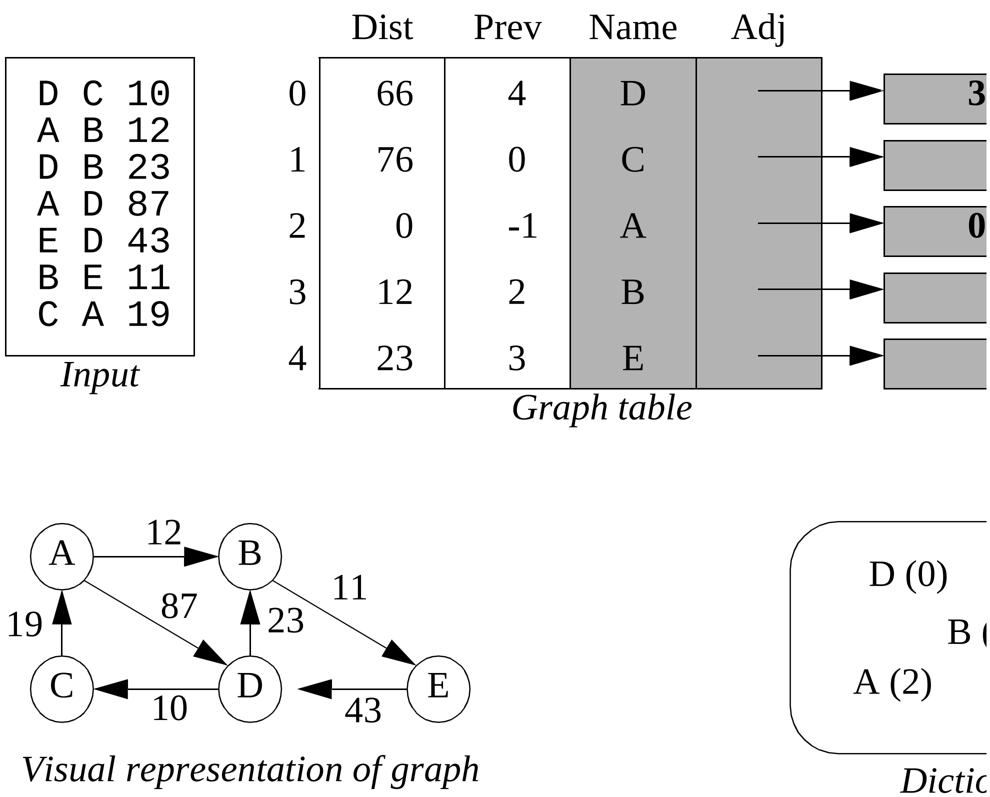
A directed graph



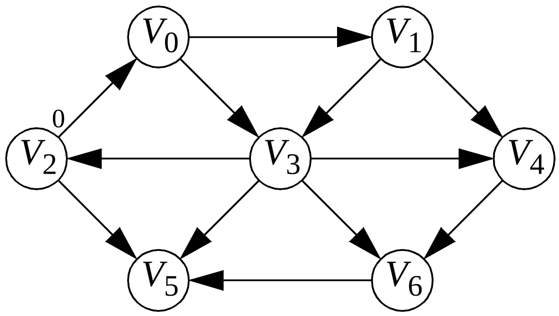
Adjacency list representation of graph in Figure 14.1; nodes in list *i* represent vertices adjacent to *i* and the cost of the connecting edge

- `Dist`: The length of the shortest path (either weighted or unweighted, depending on the algorithm) from the starting vertex to this vertex. This value is computed by the shortest path algorithm.
- `Prev`: The previous vertex on the shortest path to this vertex.
- `Name`: The name corresponding to this vertex. This is established when the vertex is placed into the dictionary and will never change. None of the shortest path algorithms examine this member. It is only used to print a final path.
- `Adj`: A pointer to a list of adjacent vertices. This is established when the graph is read. None of the shortest path algorithms will change the pointer or the linked list.

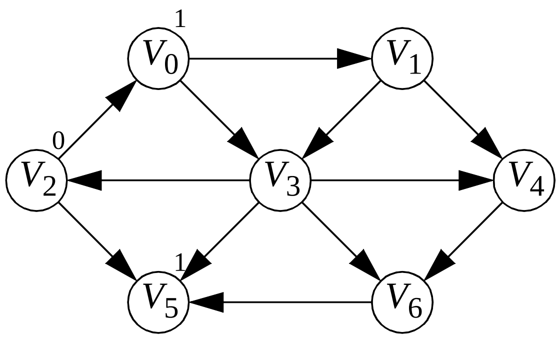
Information maintained by the Graph table



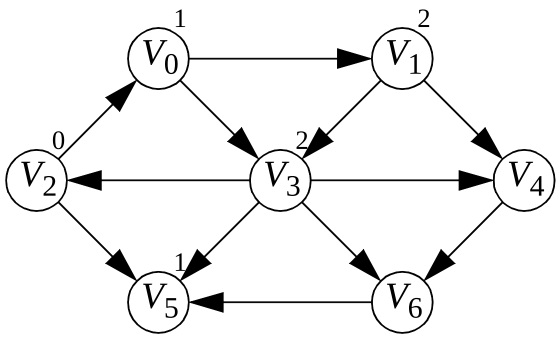
Data structures used in a shortest path calculation, with input graph taken from a file: shortest weighted path from A to C is: A to B to E to D to C (cost 76)



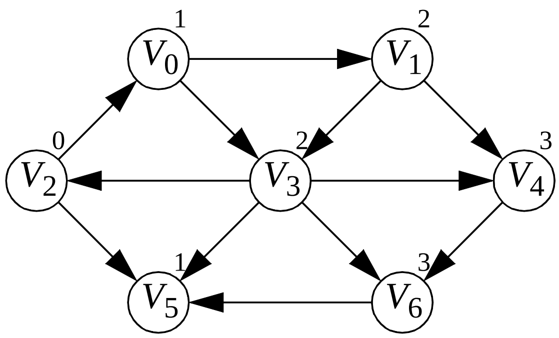
Graph after marking the start node as reachable in zero edges



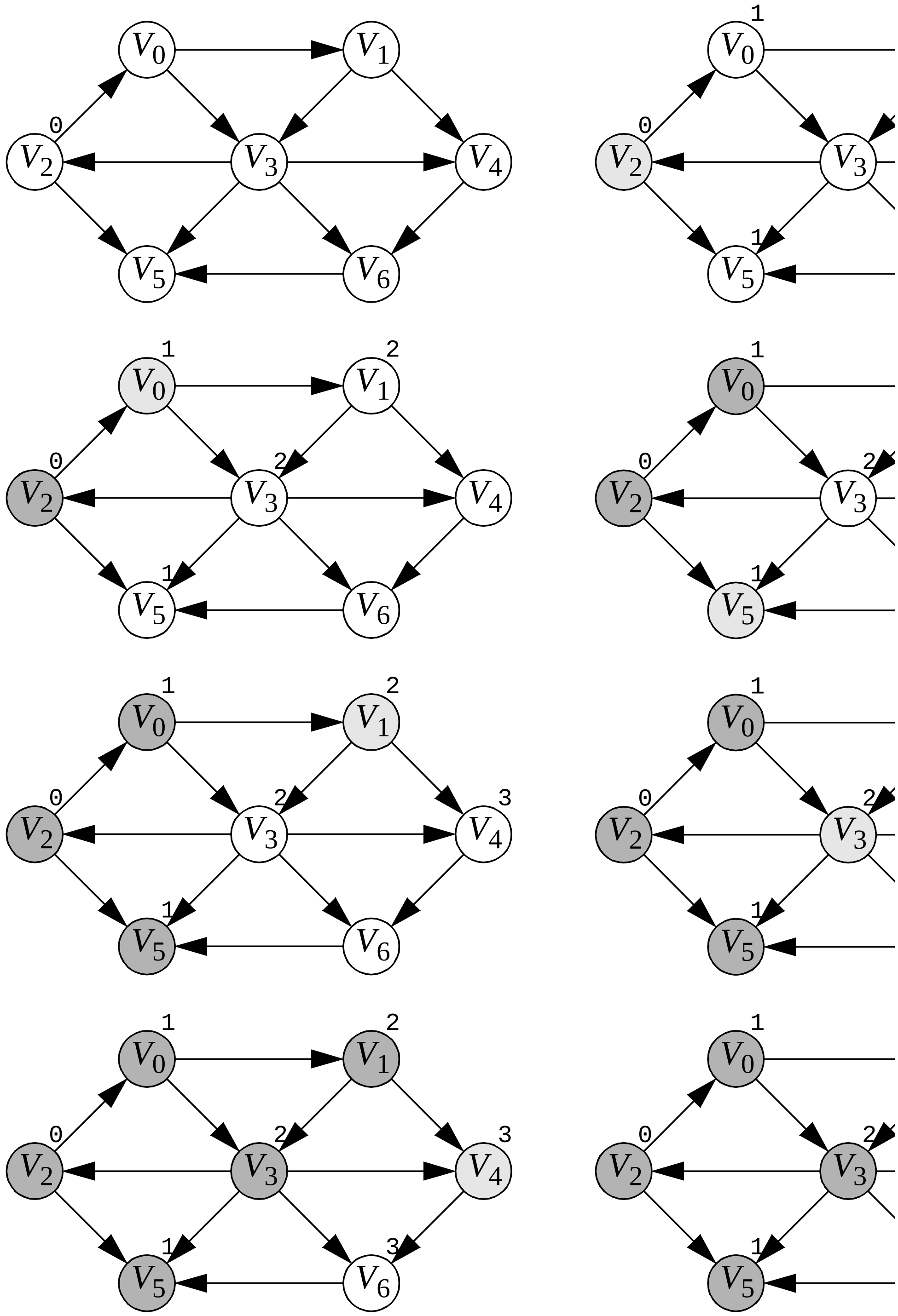
Graph after finding all vertices whose path length from the start is 1



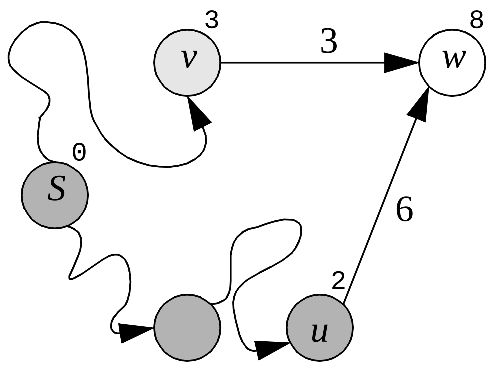
Graph after finding all vertices whose shortest path from the start is 2



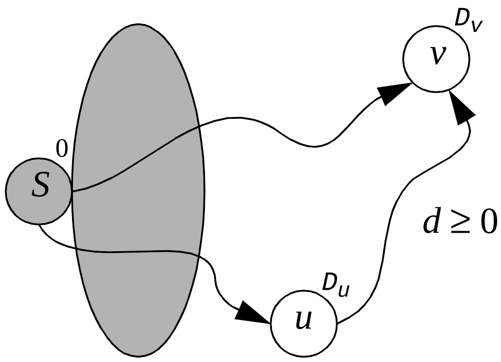
Final shortest paths



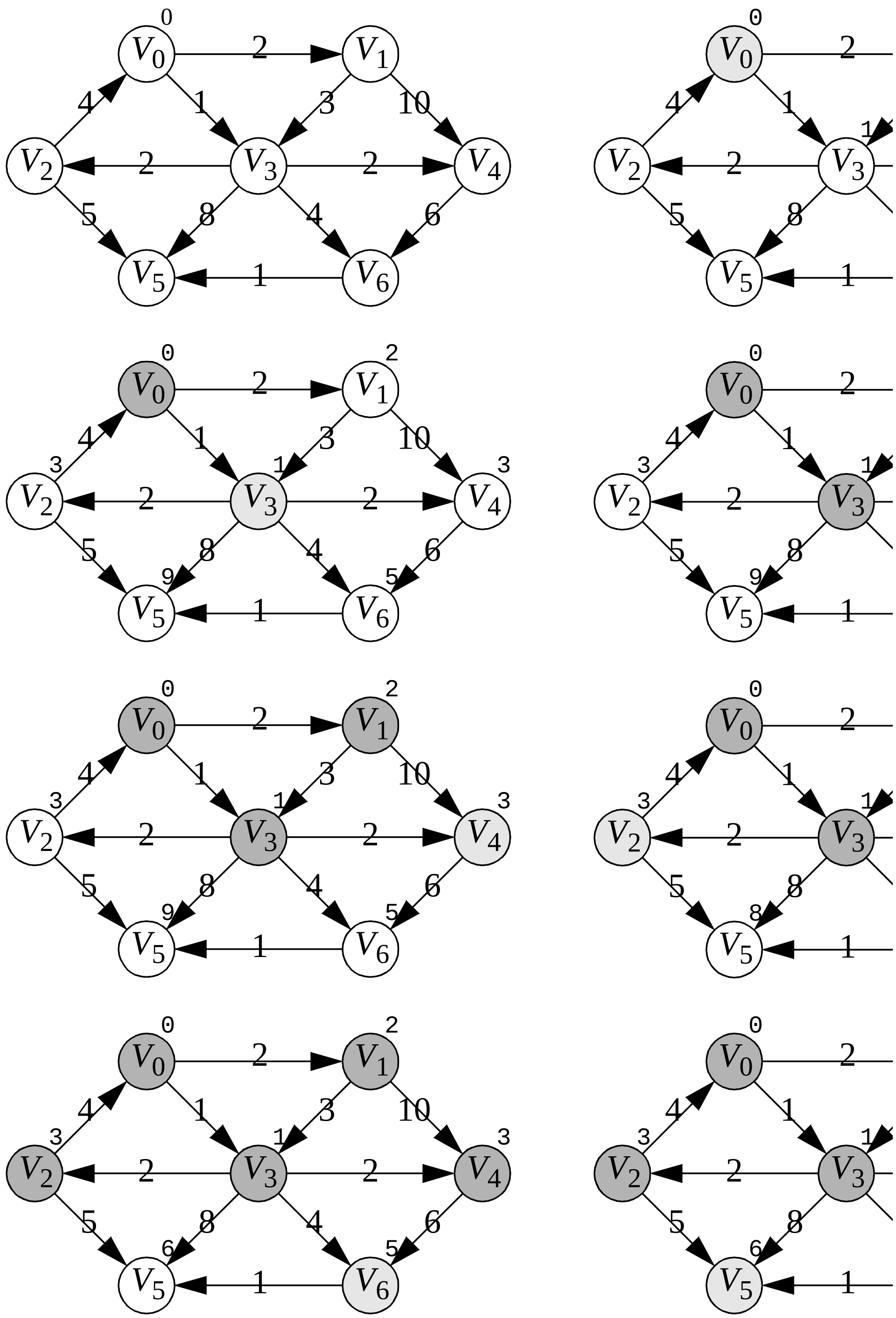
How the graph is searched in unweighted shortest path computation



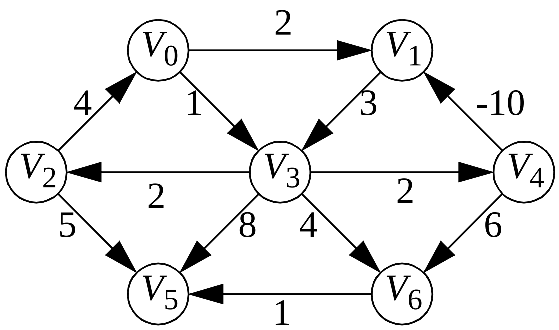
Eyeball is at v; w is adjacent; D_w should be lowered to 6



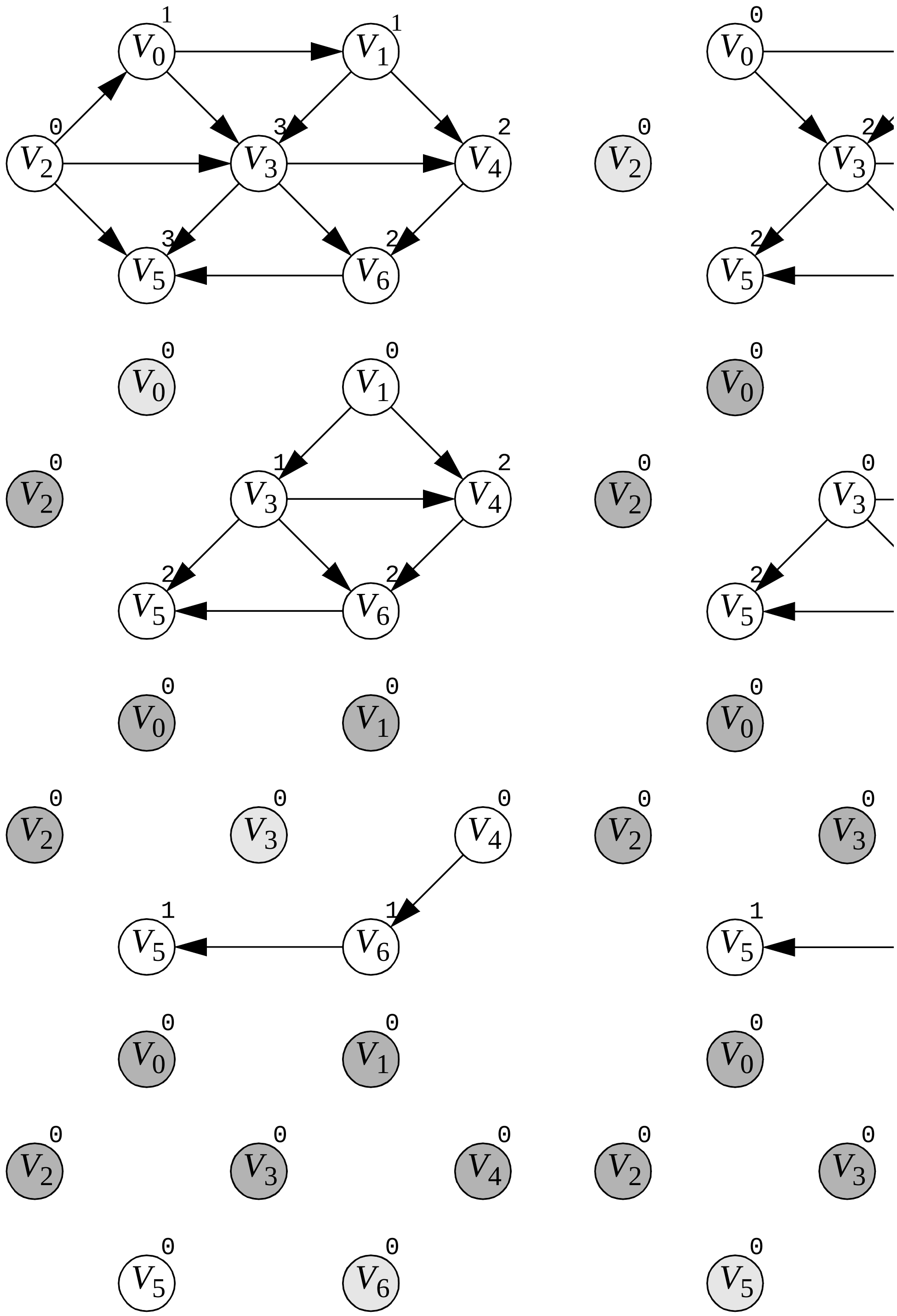
If D_v is minimal among all unseen vertices and all edge costs are nonnegative, then it represents the shortest path



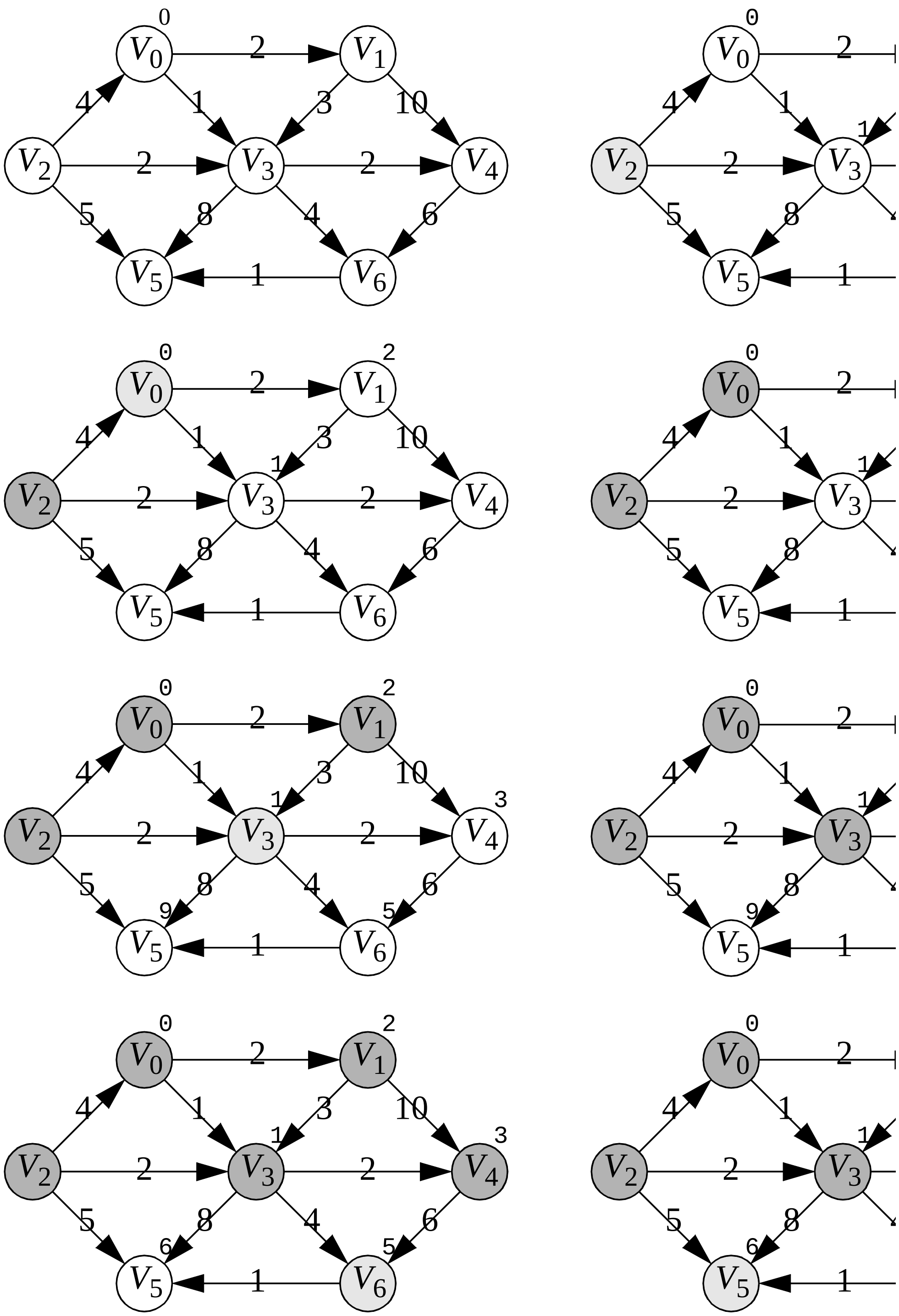
Stages of Dijkstra’s algorithm



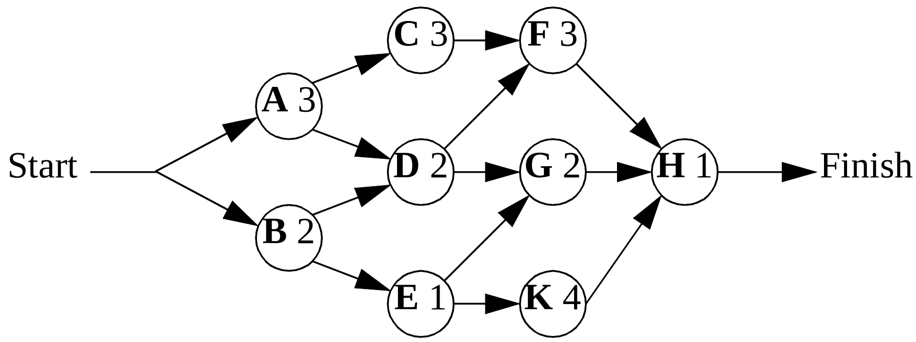
Graph with negative cost cycle



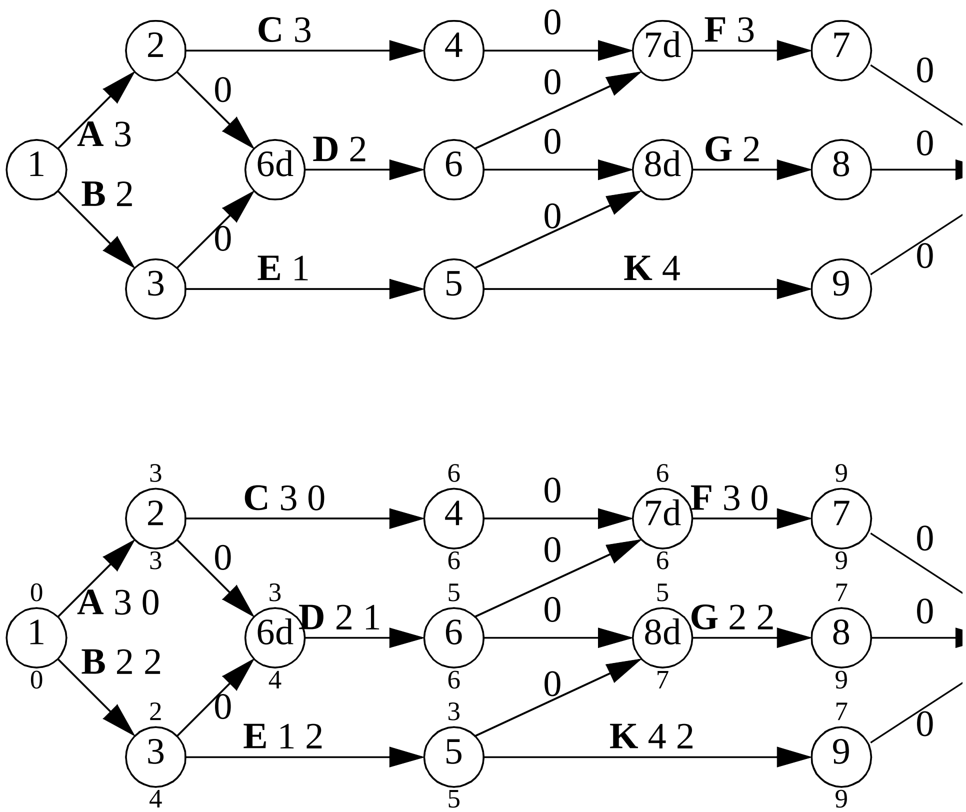
Topological sort



Stages of acyclic graph algorithm



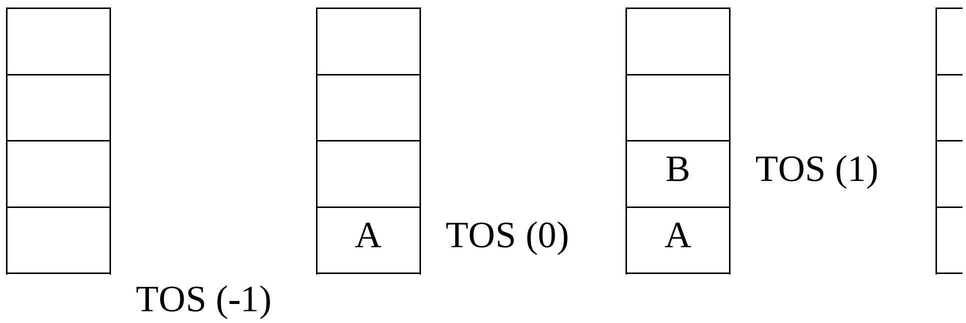
Activity-node graph



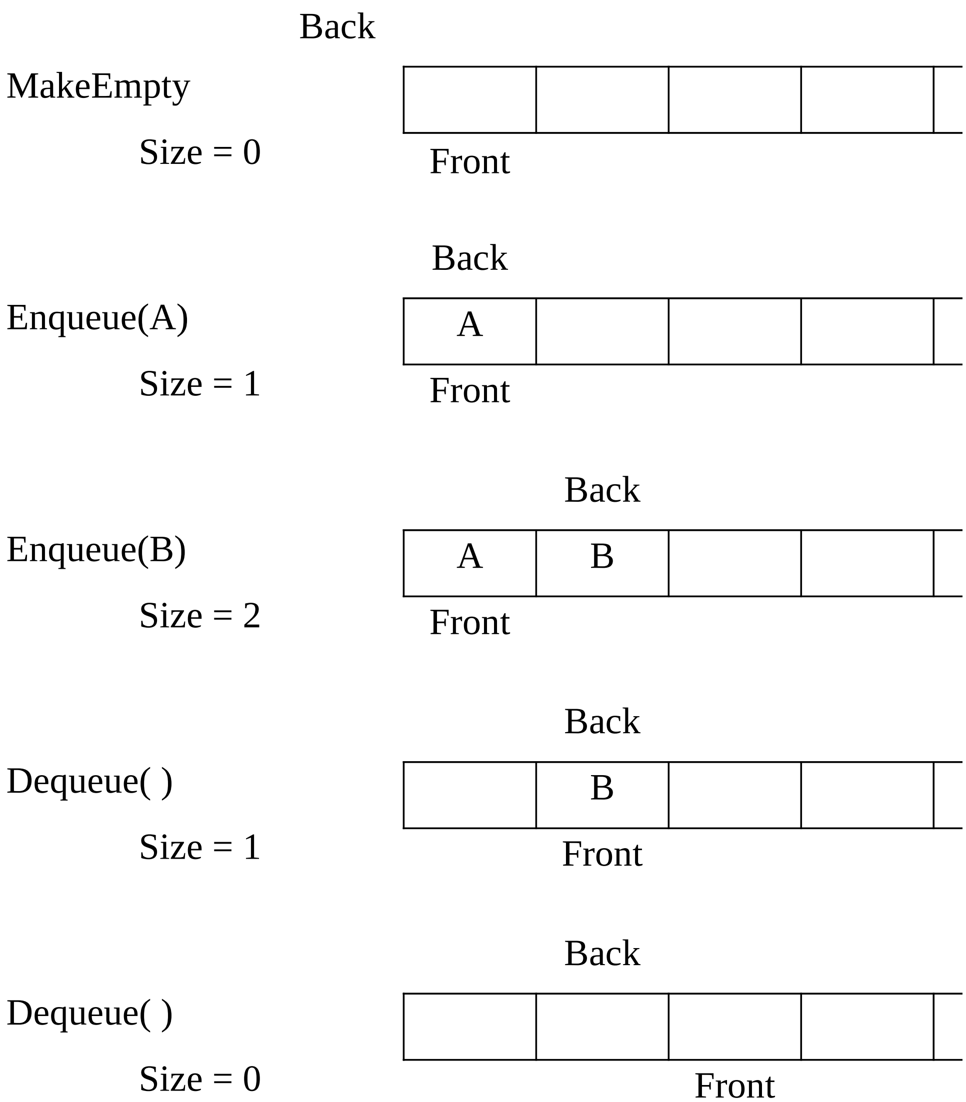
Top: Event node grap; Bottom: Earliest completion time, latest completion time, and slack (additional edge item)

Chapter 15

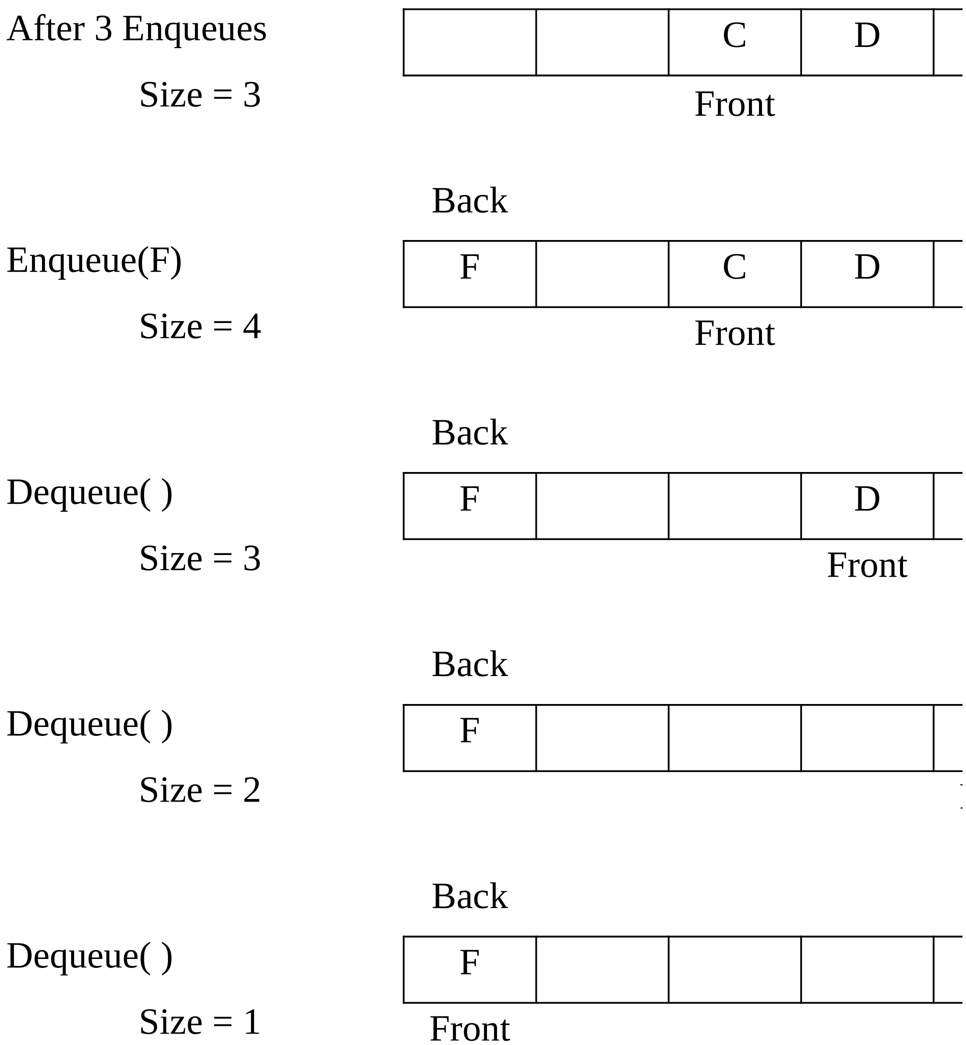
Stacks and Queues



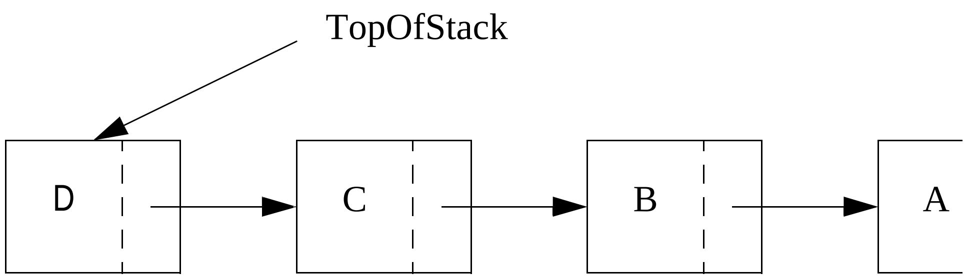
How the stack routines work: empty stack, `Push(A)`, `Push(B)`, `Pop`



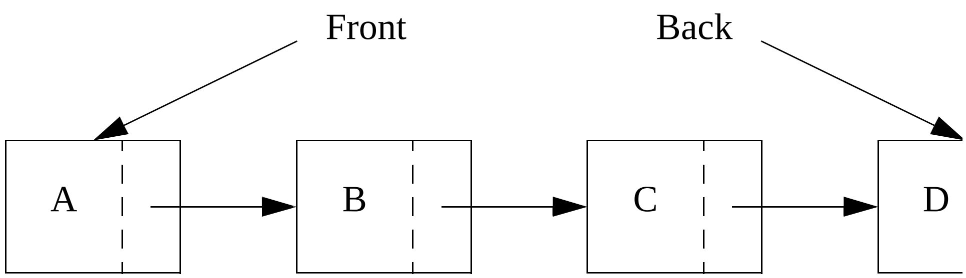
Basic array implementation of the queue



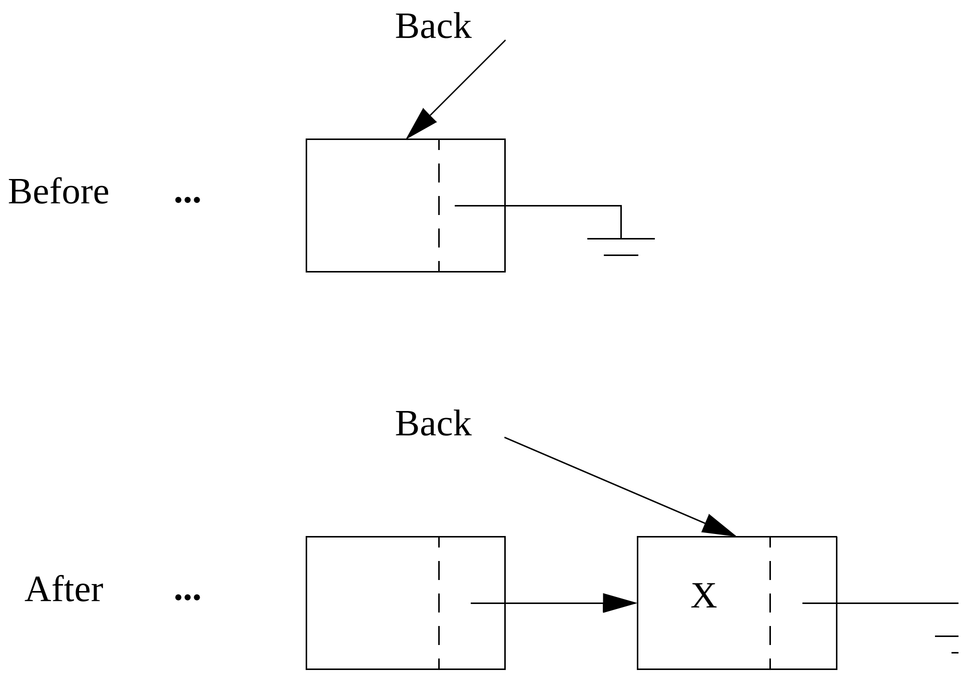
Array implementation of the queue with wraparound



Linked list implementation of the stack



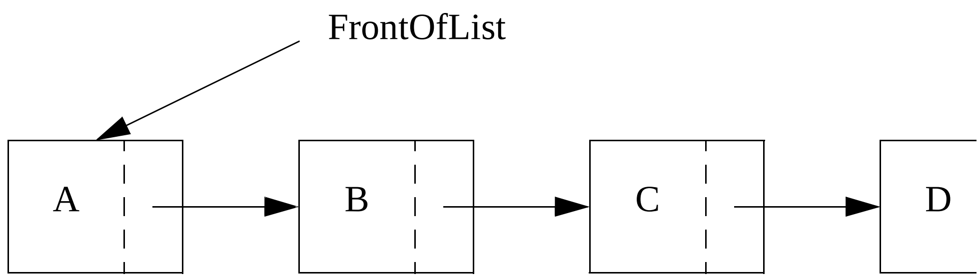
Linked list implementation of the queue



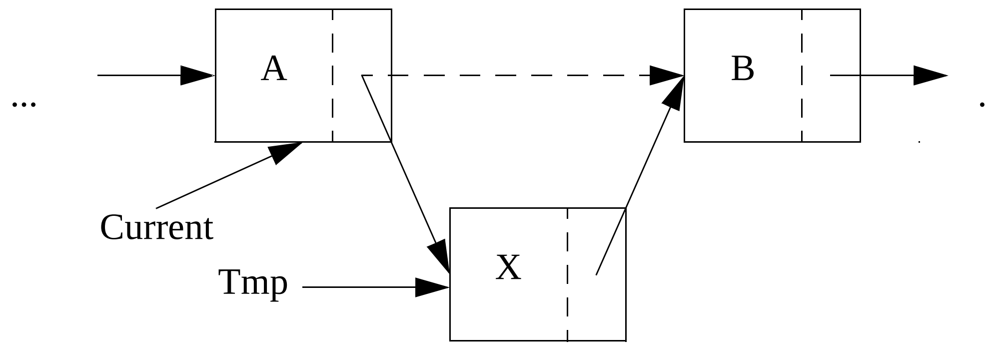
Enqueue operation for linked-list-based implementation

Chapter 16

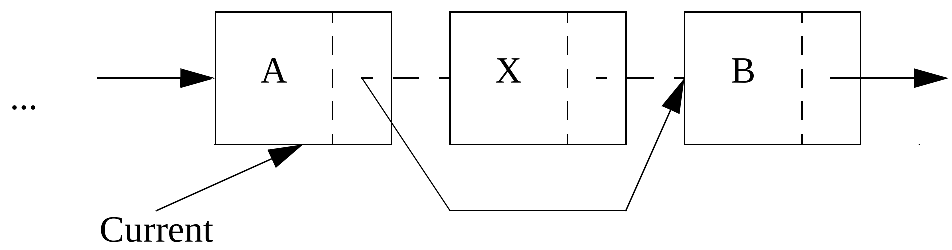
Linked Lists



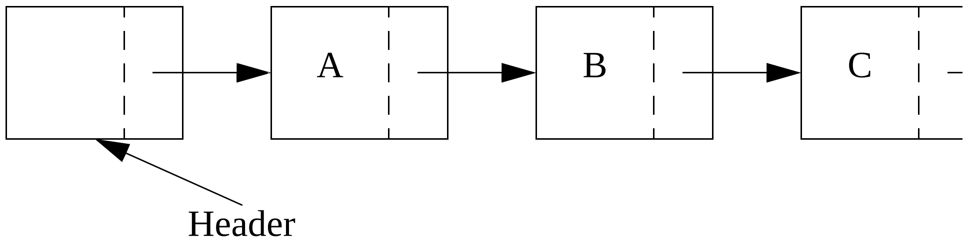
Basic linked list



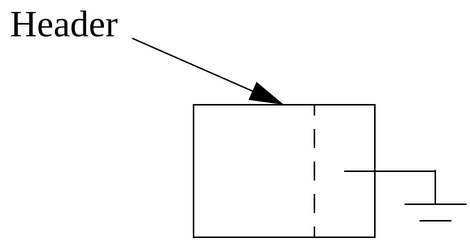
Insertion into a linked list: create new node (`Tmp`), copy in `X`, set `Tmp`'s next pointer, set `Current`'s next pointer



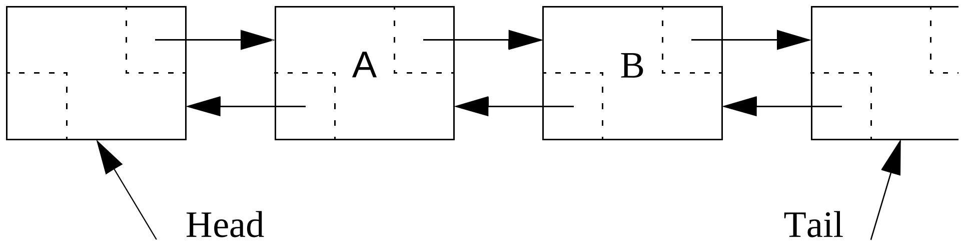
Deletion from a linked list



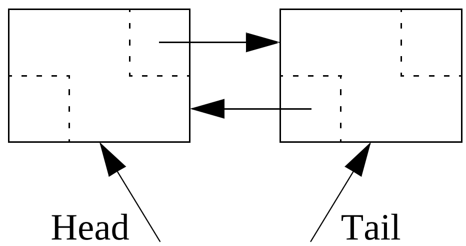
Using a header node for the linked list



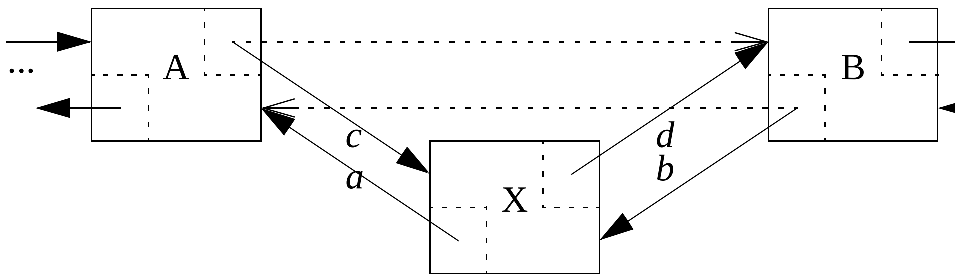
Empty list when header node is used



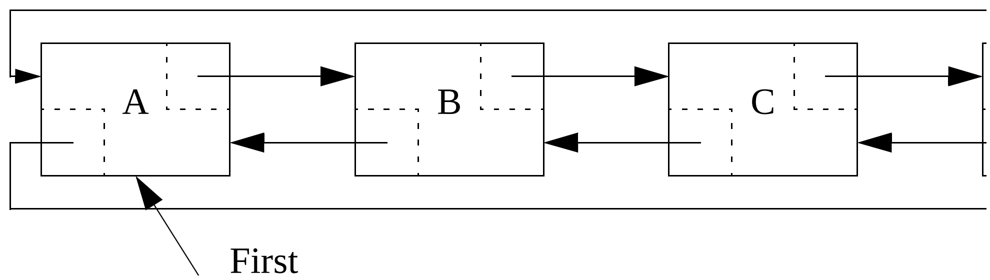
Doubly linked list



Empty doubly linked list



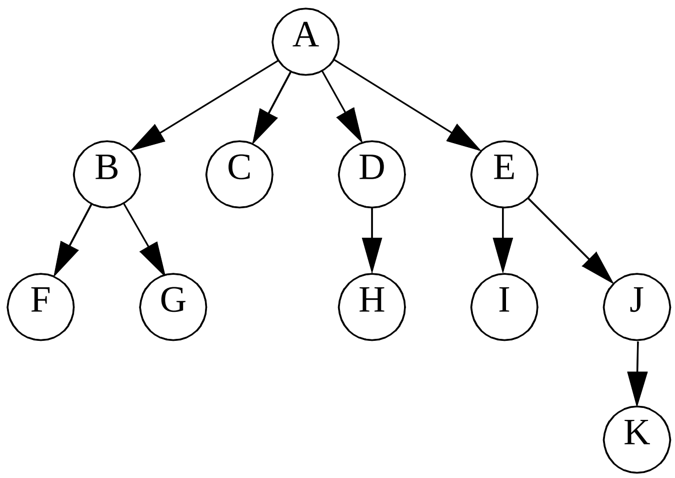
Insertion into a doubly linked list by getting new node and then changing pointers in order indicated



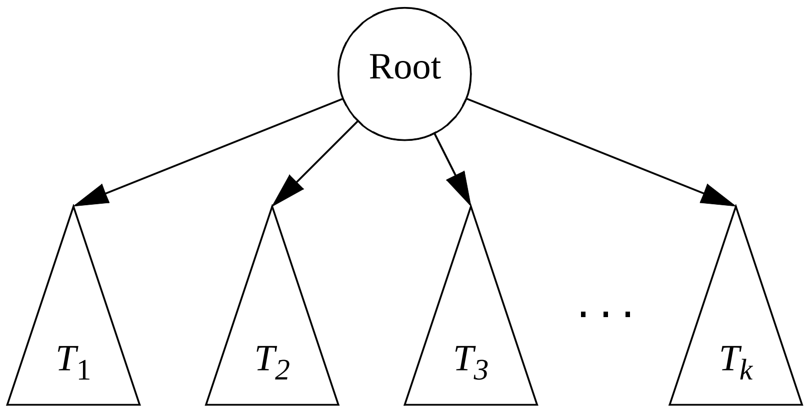
Circular doubly linked list

Chapter 17

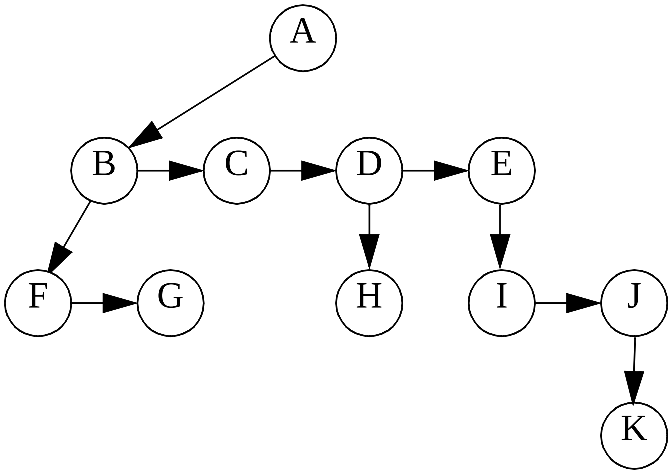
Trees



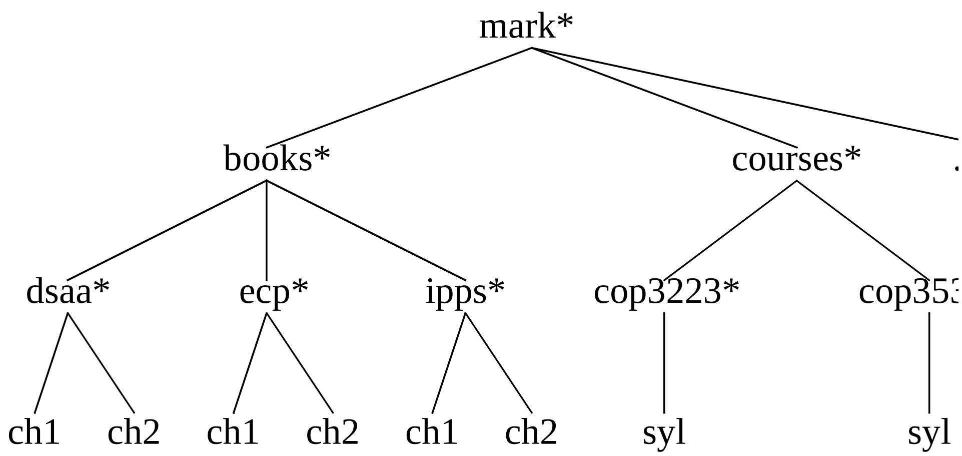
A tree



Tree viewed recursively



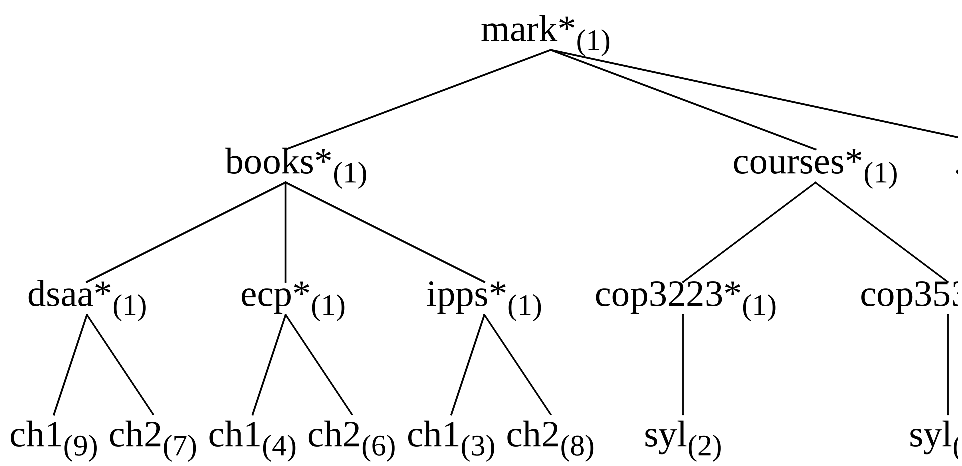
First child/next sibling representation of tree in Figure 17.1



UNIX directory

```
mark
        books
                dsaa
                        ch1
                        ch2
                ecp
                        ch1
                        ch2
                ipps
                        ch1
                        ch2
        courses
                cop3223
                        syl
                cop3530
                        syl
        .login
```

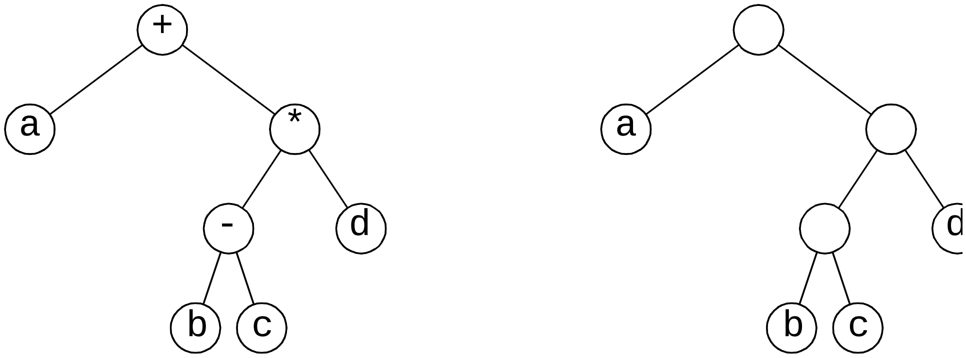
The directory listing for tree in Figure 17.4



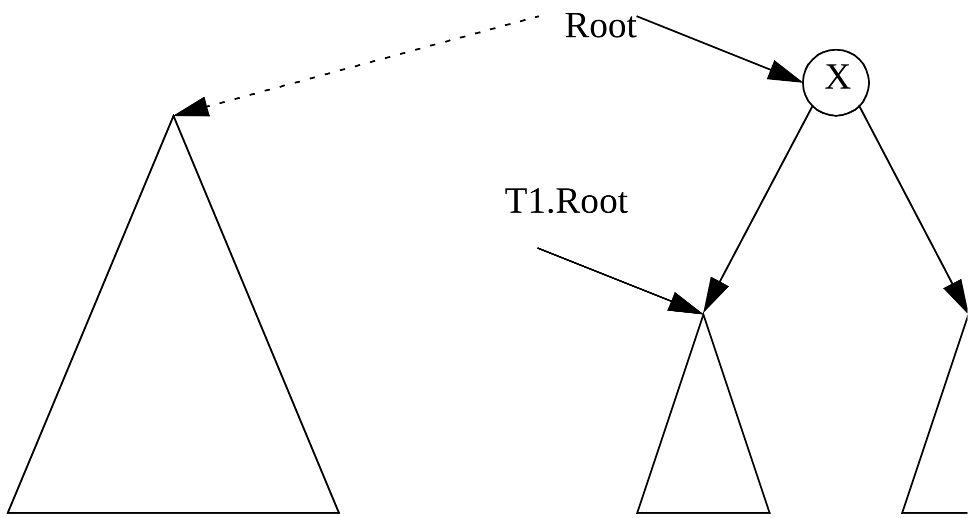
UNIX directory with file sizes

```
                        ch1                          9
                        ch2                          7
                dsaa                                17
                        ch1                          4
                        ch2                          6
                ecp                                 11
                        ch1                          3
                        ch2                          8
                ipps                                12
        books                                       41
                        syl                          2
                cop3223                              3
                        syl                          3
                cop3530                              4
        courses                                      8
        .login                                       2
mark                                                52
```

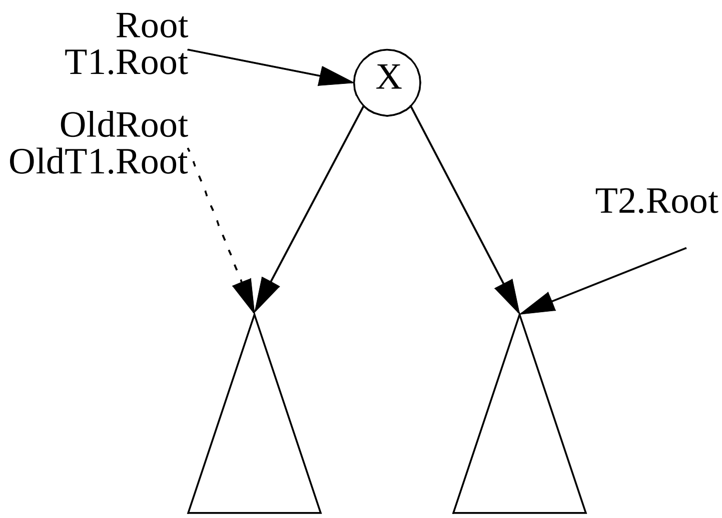
Trace of the `Size` function



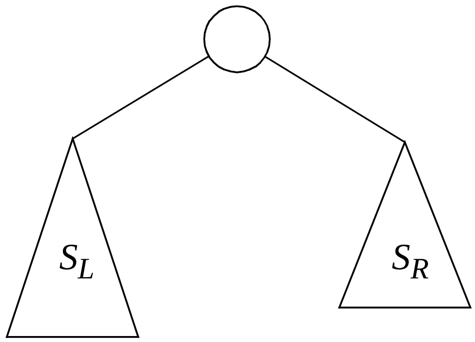
Uses of binary trees: left is an expression tree and right is a Huffman coding tree



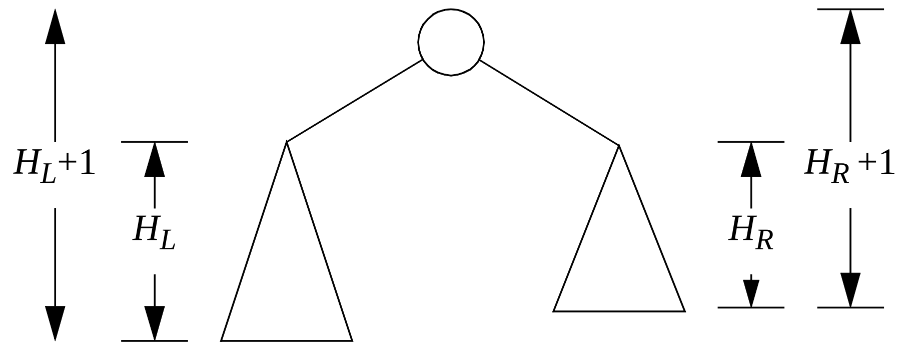
Result of a naive `Merge` operation



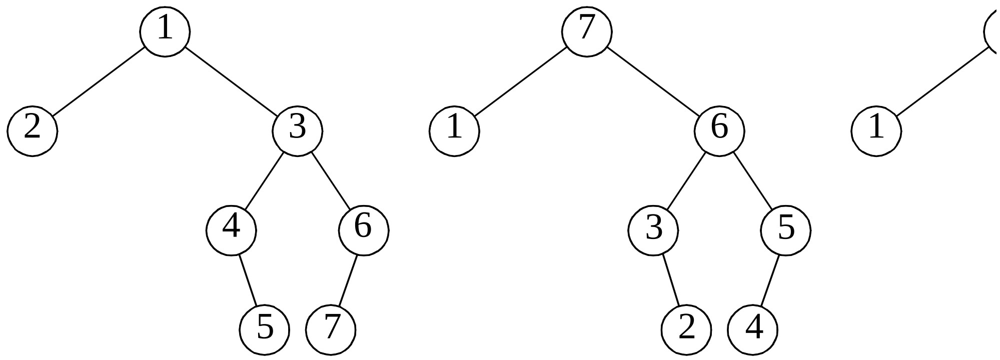
Aliasing problems in the `Merge` operation; `T1` is also the current object



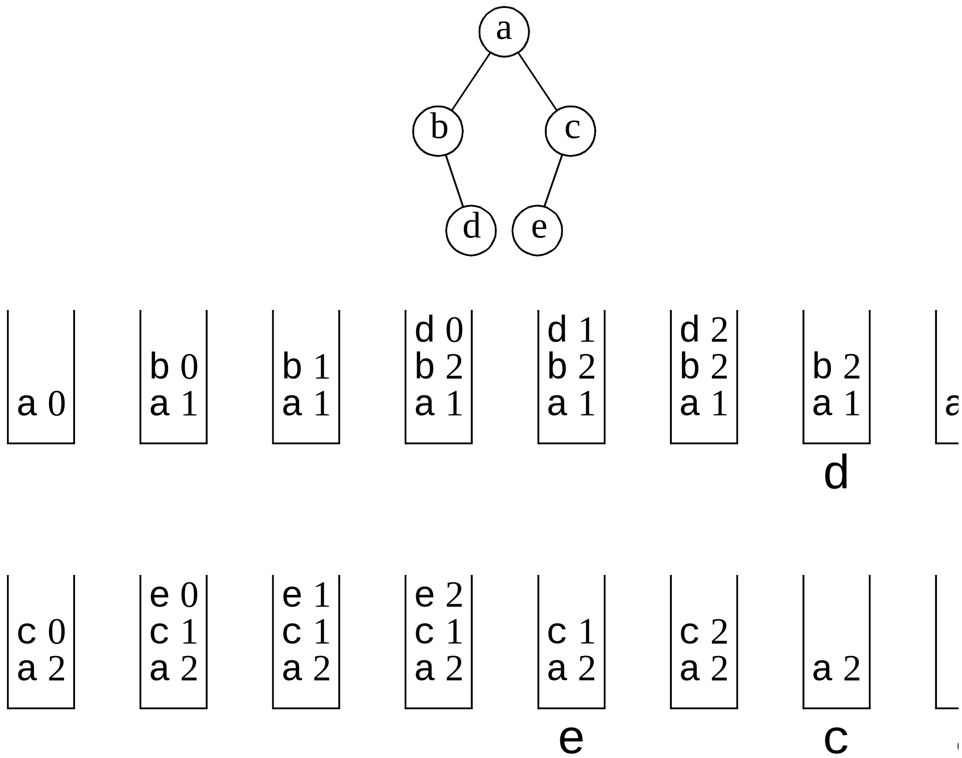
Recursive view used to calculate the size of a tree: $S_T = S_L + S_R + 1$



Recursive view of node height calculation: H_T = Max(H_L+1, H_R +1)



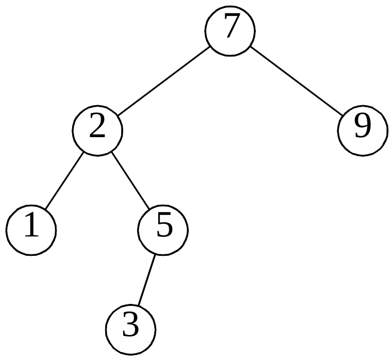
Preorder, postorder, and inorder visitation routes

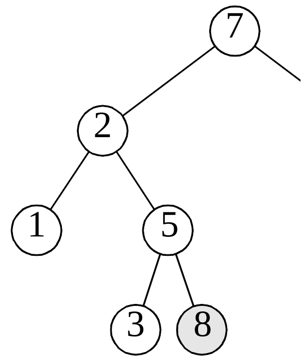


Stack states during postorder traversal

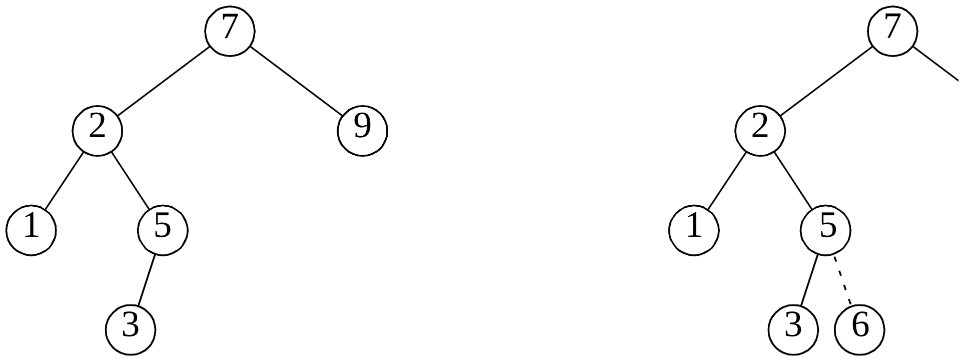
Chapter 18

Binary Search Trees

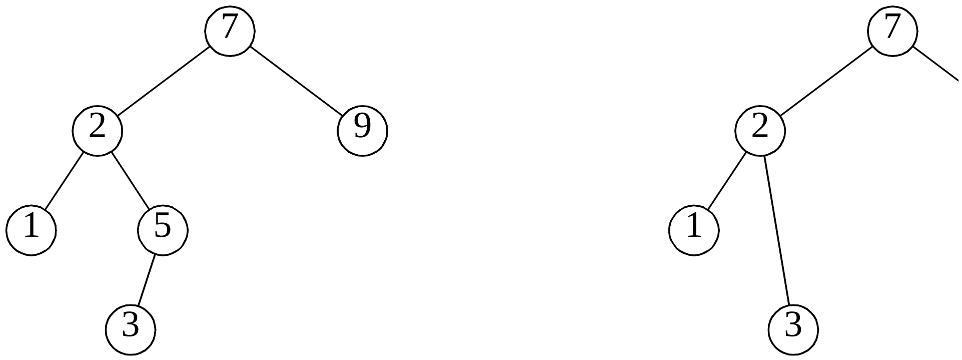




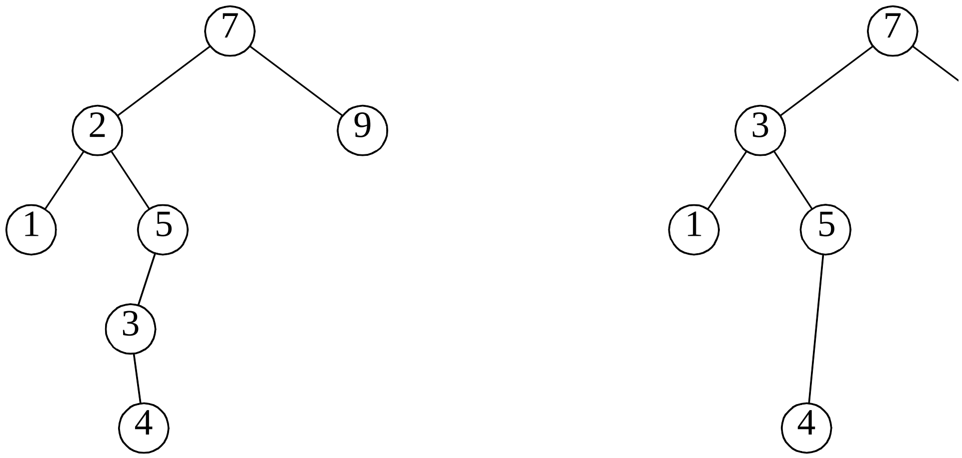
Two binary trees (only the left tree is a search tree)



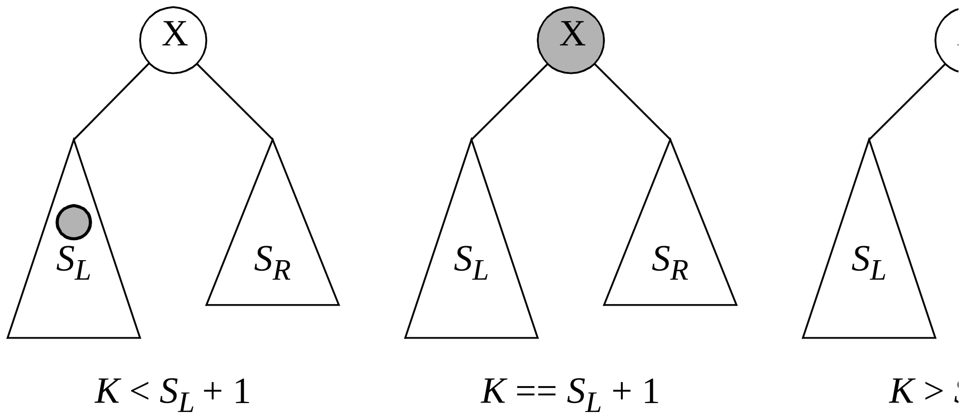
Binary search trees before and after inserting 6



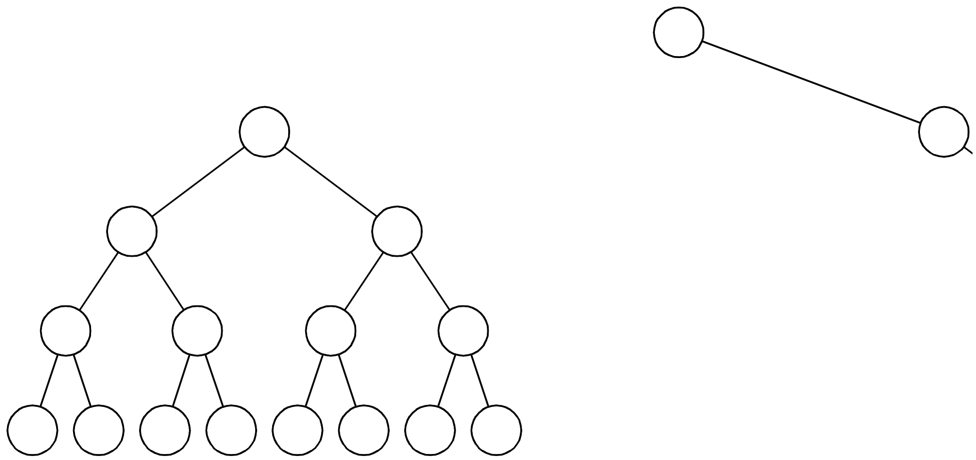
Deletion of node 5 with one child, before and after



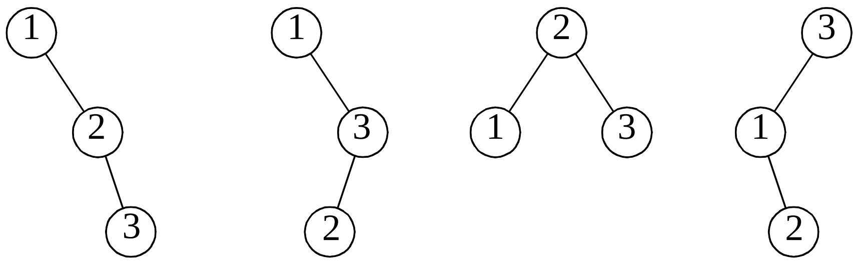
Deletion of node 2 with two children, before and after



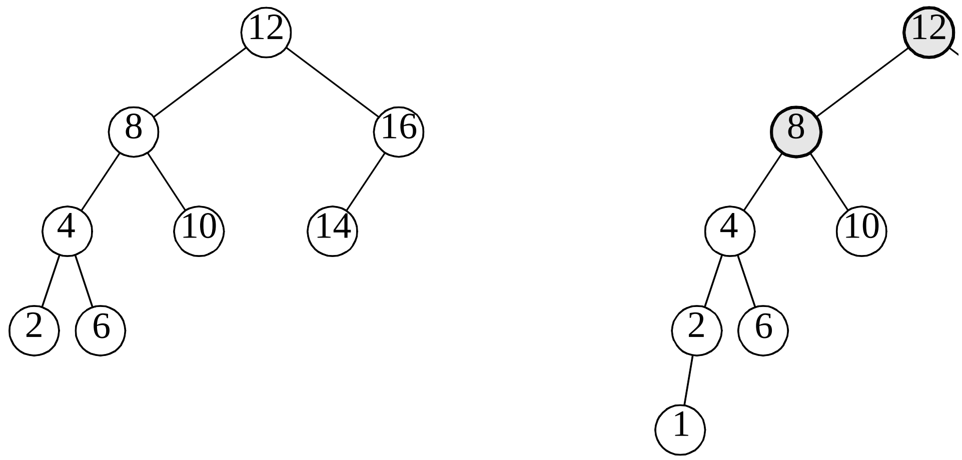
Using the `Size` data member to implement `FindKth`



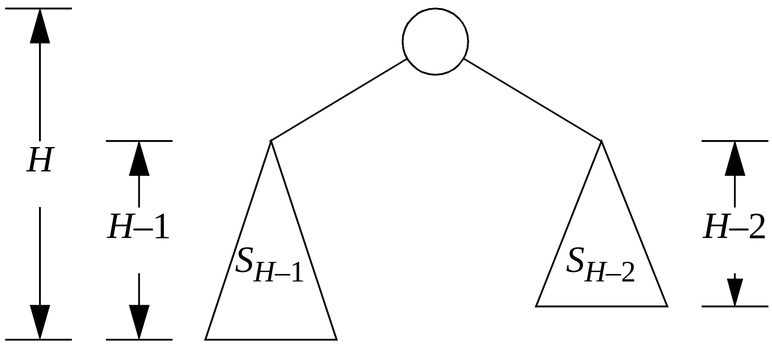
Balanced tree on the left has a depth of log N; unbalanced tree on the right has a depth of N–1



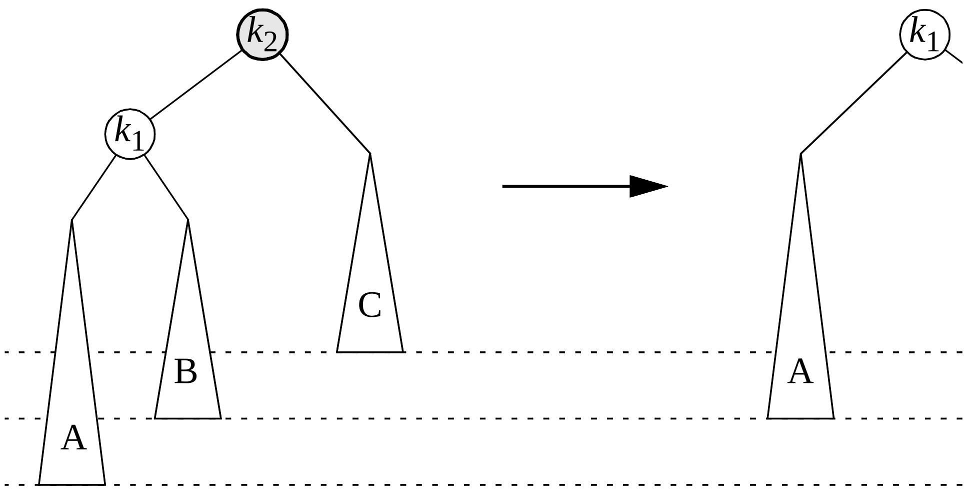
Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree in the middle is twice as likely as any other



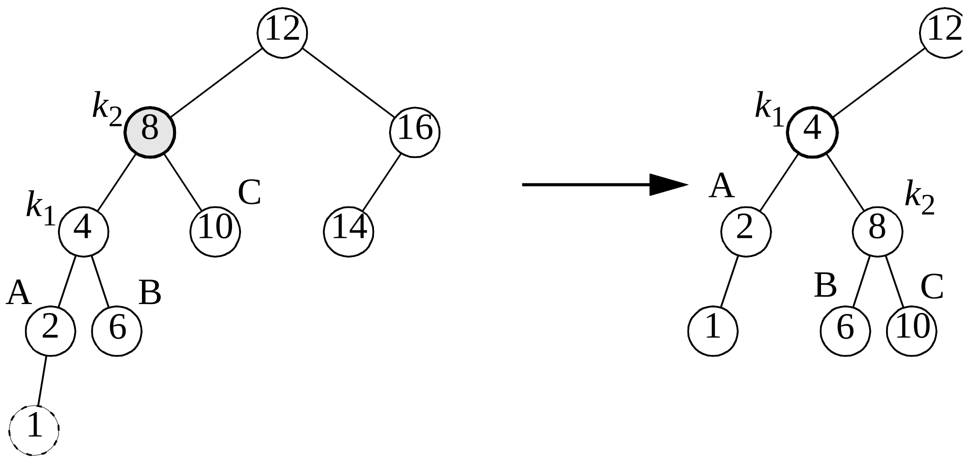
Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened)



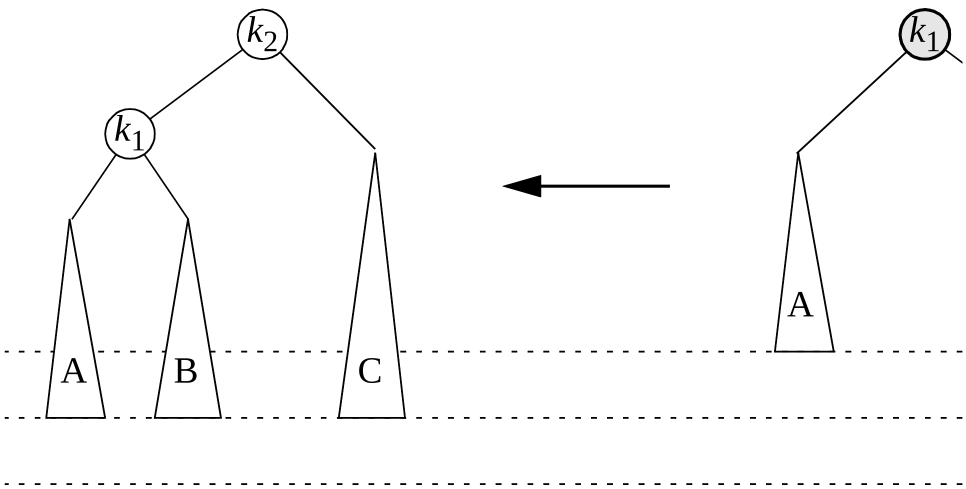
Minimum tree of height *H*



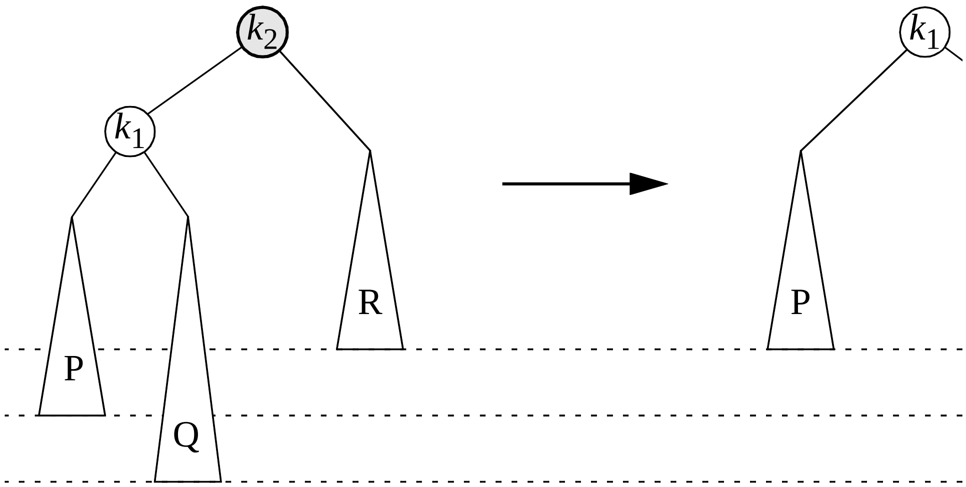
Single rotation to fix case 1



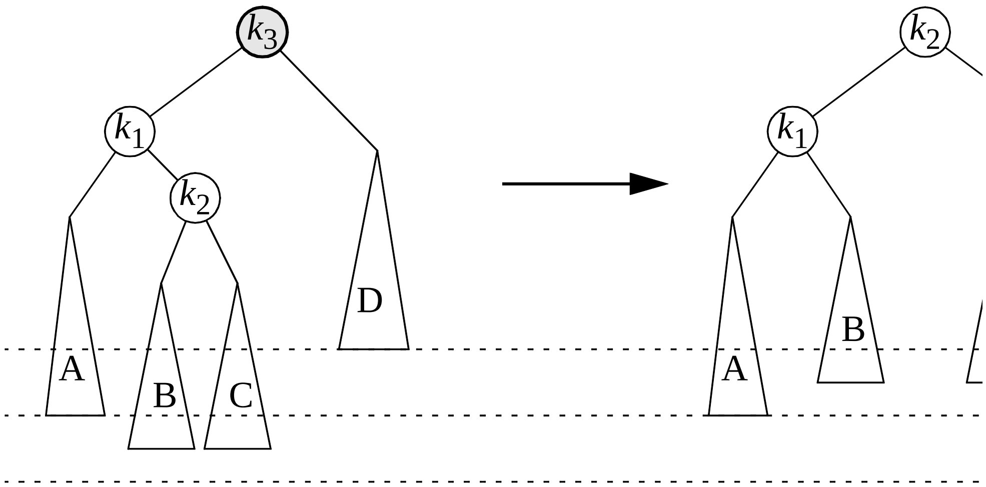
Single rotation fixes AVL tree after insertion of 1



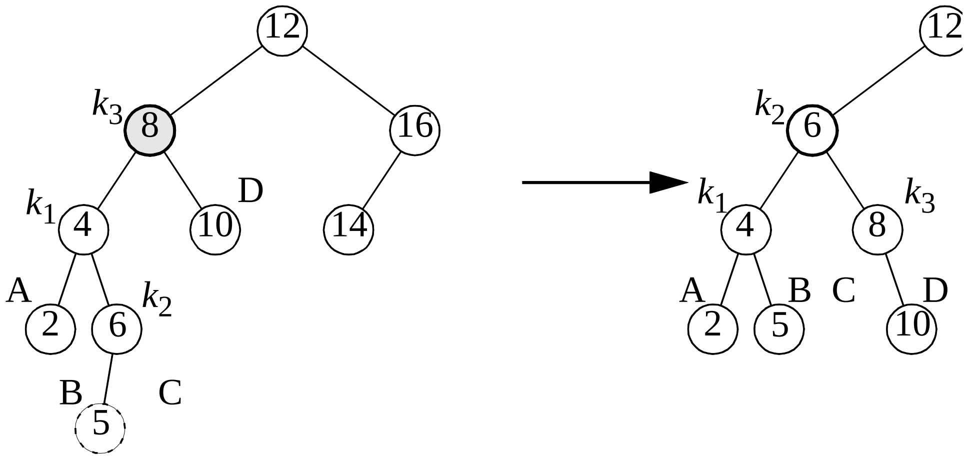
Symmetric single rotation to fix case 4



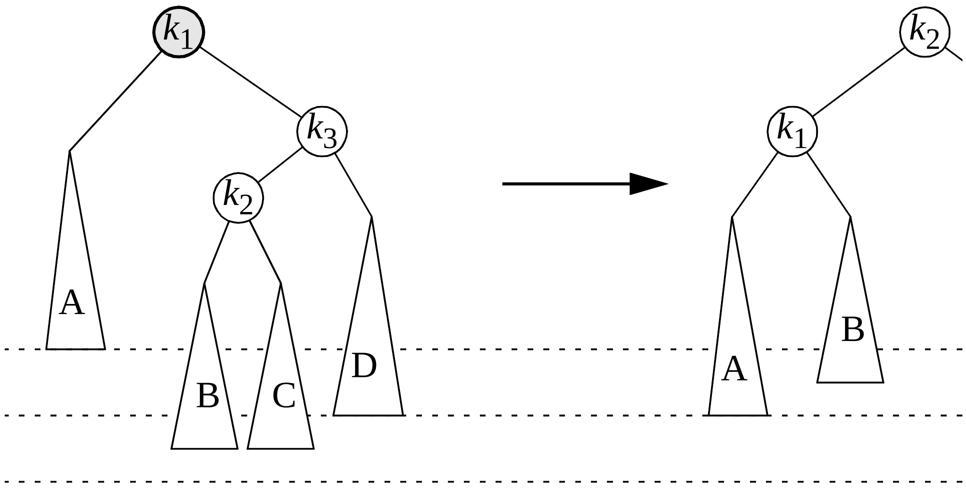
Single rotation does not fix case 2



Left-right double rotation to fix case 2



Double rotation fixes AVL tree after insertion of 5

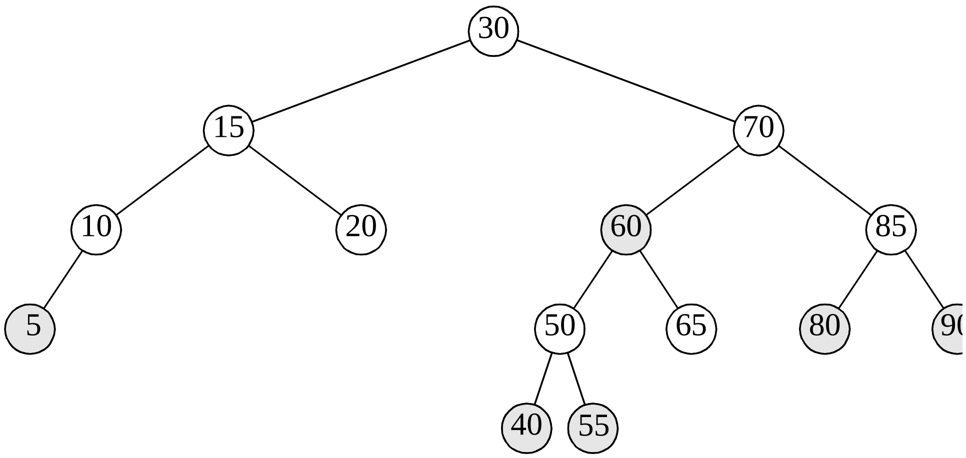


Left-right double rotation to fix case 3

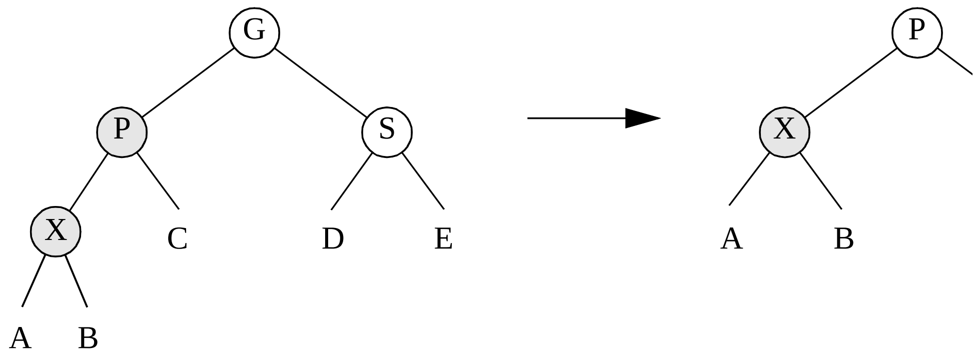
A red black tree is a binary search tree with the following ordering properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a `NULL` pointer must contain the same number of black nodes.

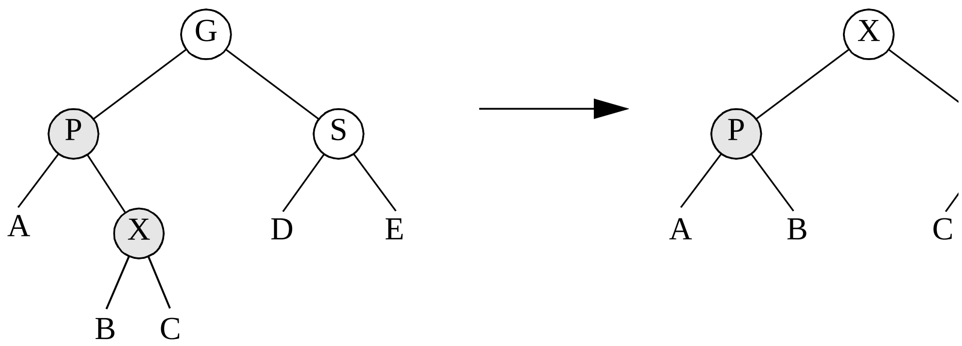
Red black tree properties



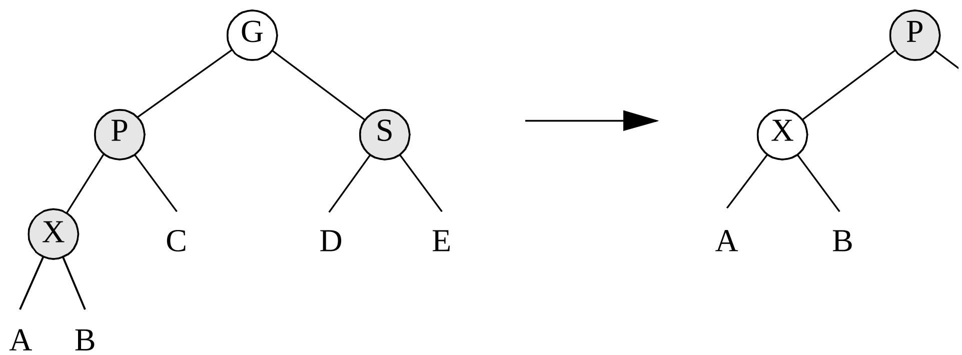
Example of a red black tree; insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)



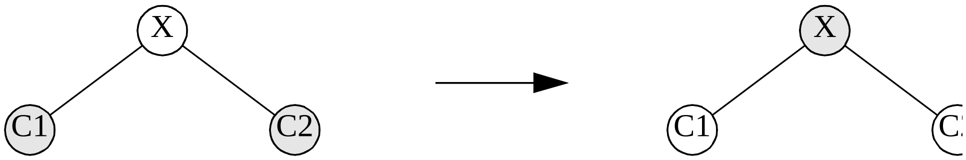
If *S* is black, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 if *X* is an outside grandchild



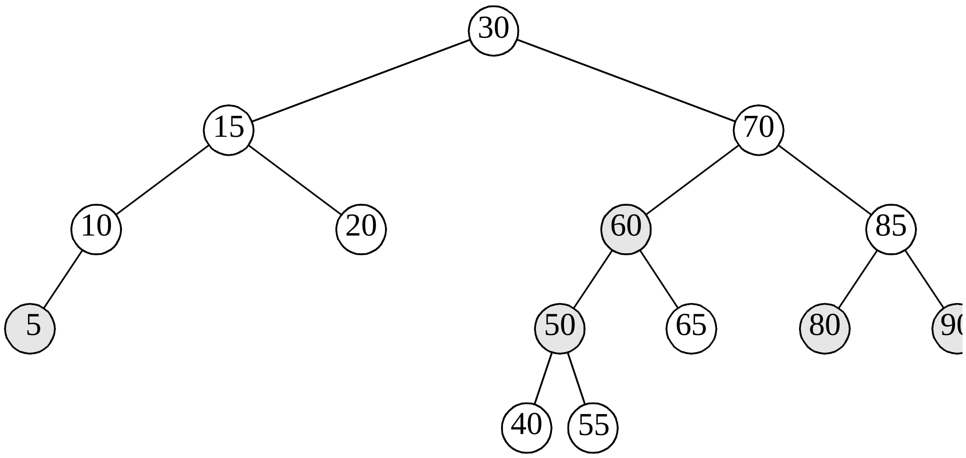
If *S* is black, then a double rotation involving *X*, the parent, and the grandparent, with appropriate color changes, restores property 3 if *X* is an inside grandchild



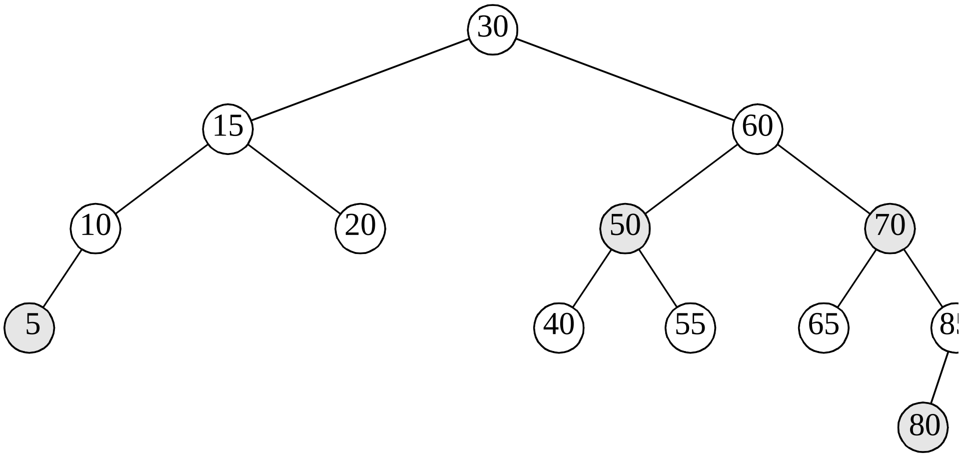
If *S* is red, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 between *X* and *P*



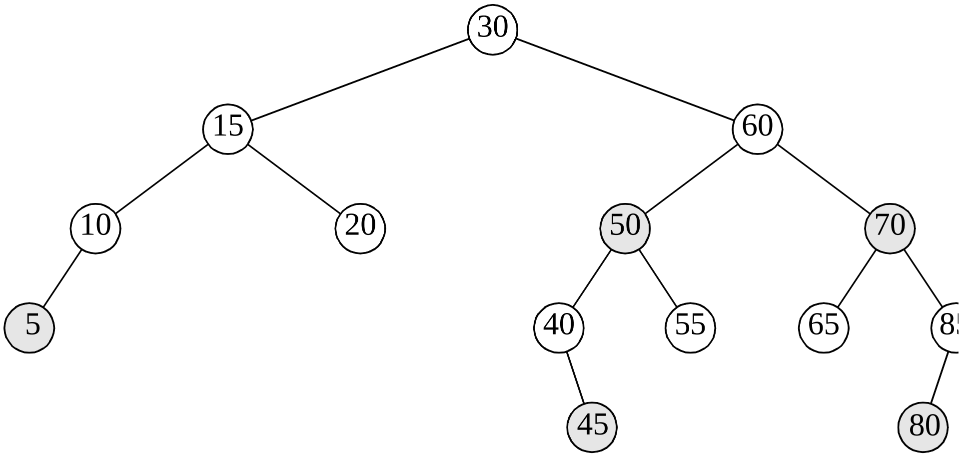
Color flip; only if *X*'s parent is red do we continue with a rotation



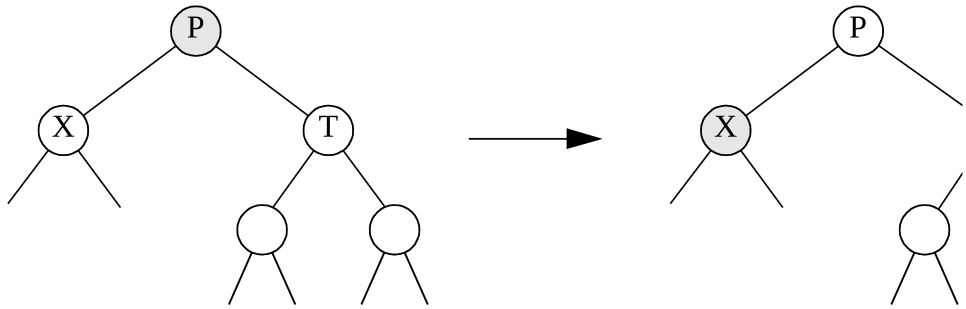
Color flip at 50 induces a violation; because it is outside, a single rotation fixes it



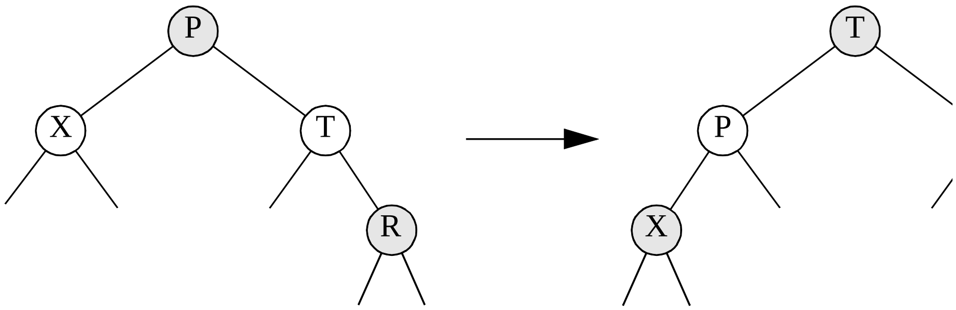
Result of single rotation that fixes violation at node 50



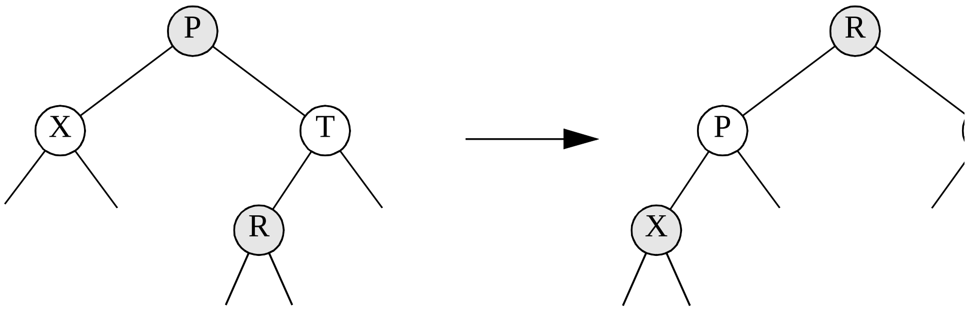
Insertion of 45 as a red node



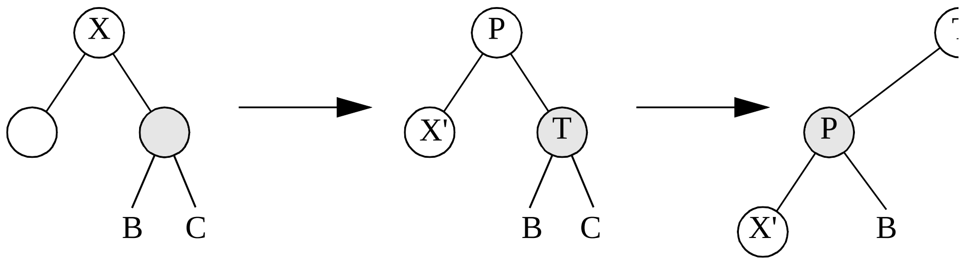
Deletion: *X* has two black children, and both of its sibling's children are black; do a color flip



Deletion: *X* has two black children, and the outer child of its sibling is red; do a single rotation



Deletion: *X* has two black children, and the inner child of its sibling is red; do a double rotation



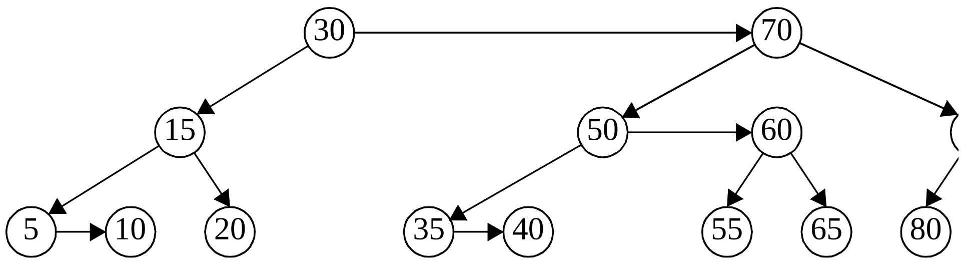
X is black and at least one child is red; if we fall through to next level and land on a red child, everything is good; if not, we rotate a sibling and parent

The level of a node is

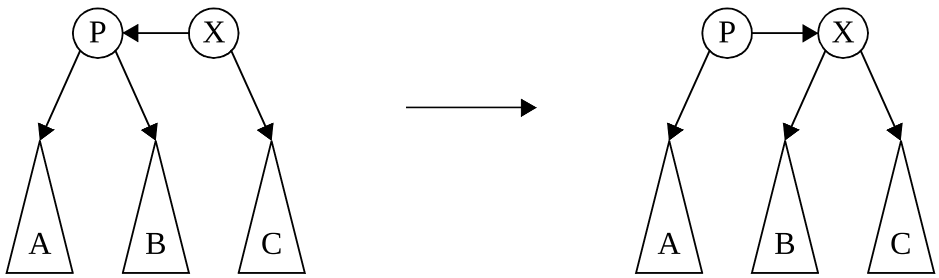
- One if the node is a leaf
- The level of its parent, if the node is red
- One less than the level of its parent, if the node is black

1. Horizontal links are right pointers (because only right children may be red).
2. There may not be two consecutive horizontal links (because there cannot be consecutive red nodes).
3. Nodes at level 2 or higher must have two children.
4. If a node does not have a right horizontal link, then its two children are at the same level.

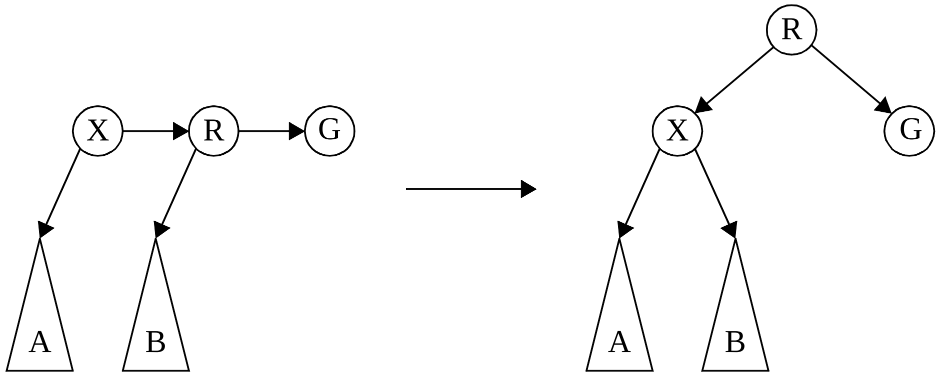
AA-tree properties



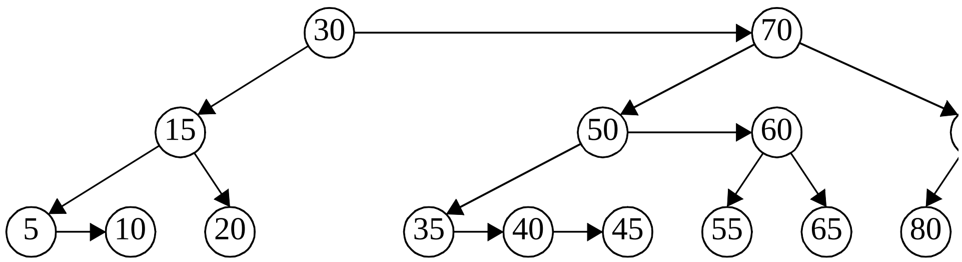
AA-tree resulting from insertion of 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, 35



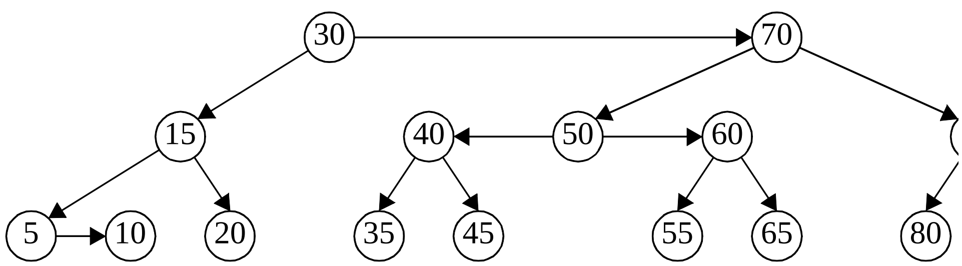
`Skew` is a simple rotation between X and P



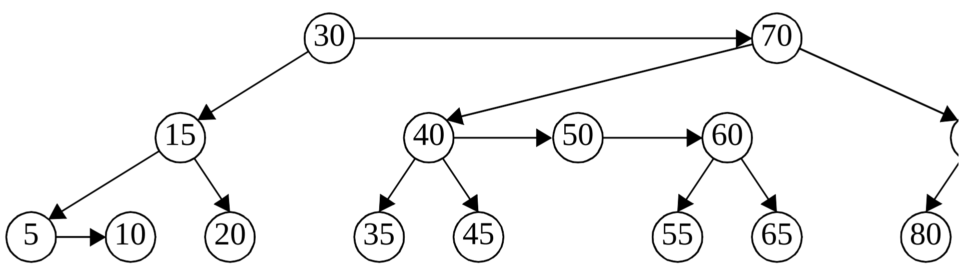
`Split` is a simple rotation between `X` and `R`; note that `R`'s level increases



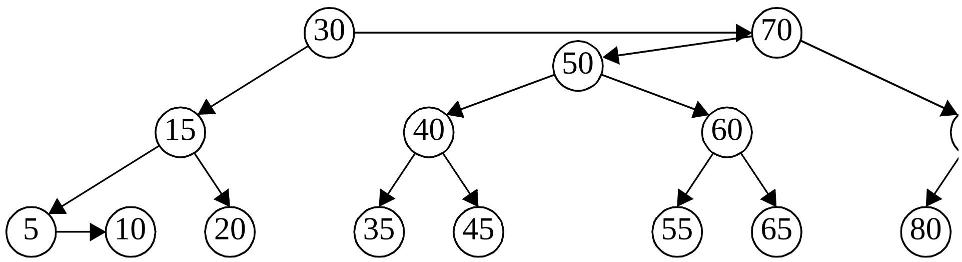
After inserting 45 into sample tree; consecutive horizontal links are introduced starting at 35



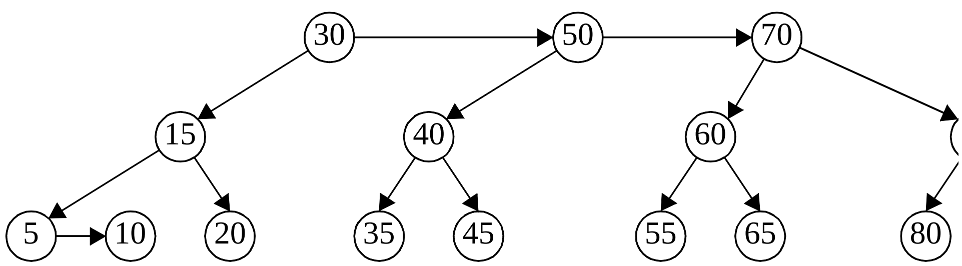
After `Split` at 35; introduces a left horizontal link at 50



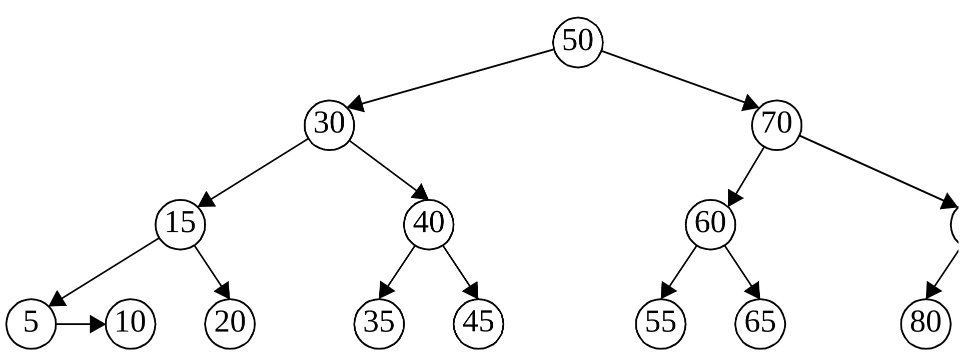
After `Skew` at 50; introduces consecutive horizontal nodes starting at 40



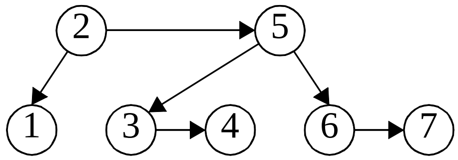
After `Split` at 40; 50 is now on the same level as 70, thus inducing an illegal left horizontal link



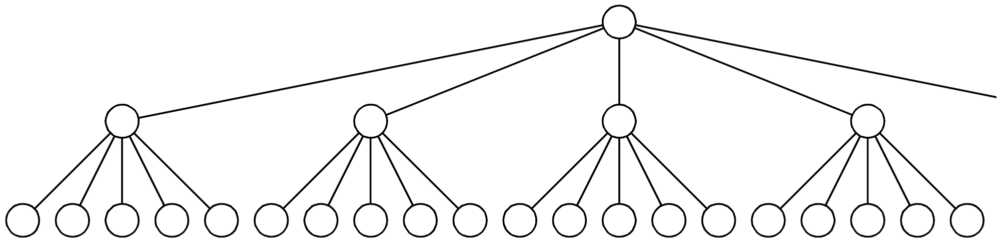
After `Skew` at 70; this introduces consecutive horizontal links at 30



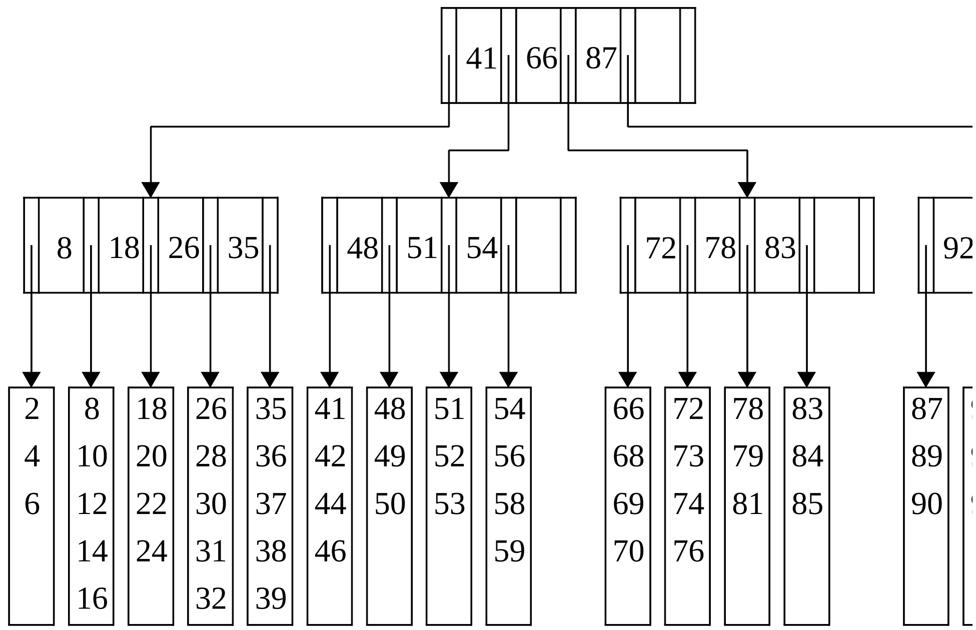
After `Split` at 30; insertion is complete



When 1 is deleted, all nodes become level 1, introducing horizontal left links



Five-ary tree of 31 nodes has only three levels

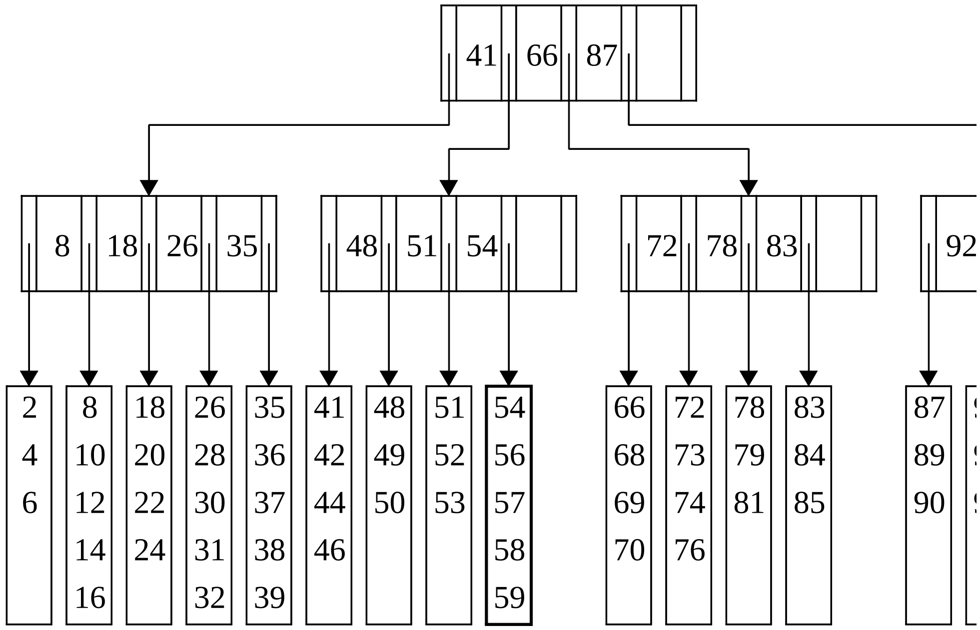


B-tree of order 5

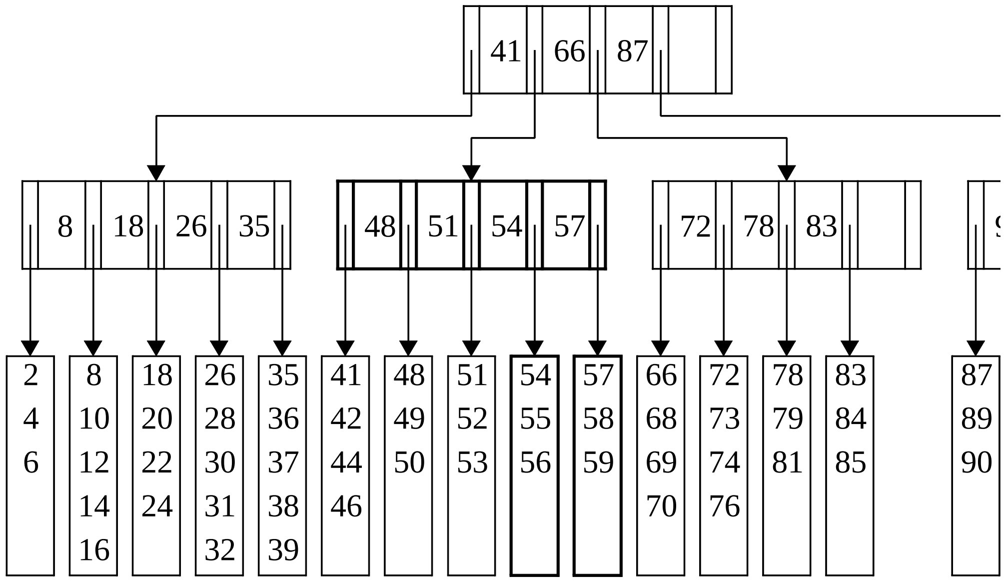
A B-tree of order M is an M-ary tree with the following properties:

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M - 1$ keys to guide the searching; key i represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between 2 and M children.
4. All nonleaf nodes (except the root) have between $\lceil M / 2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L / 2 \rceil$ and L children, for some L.

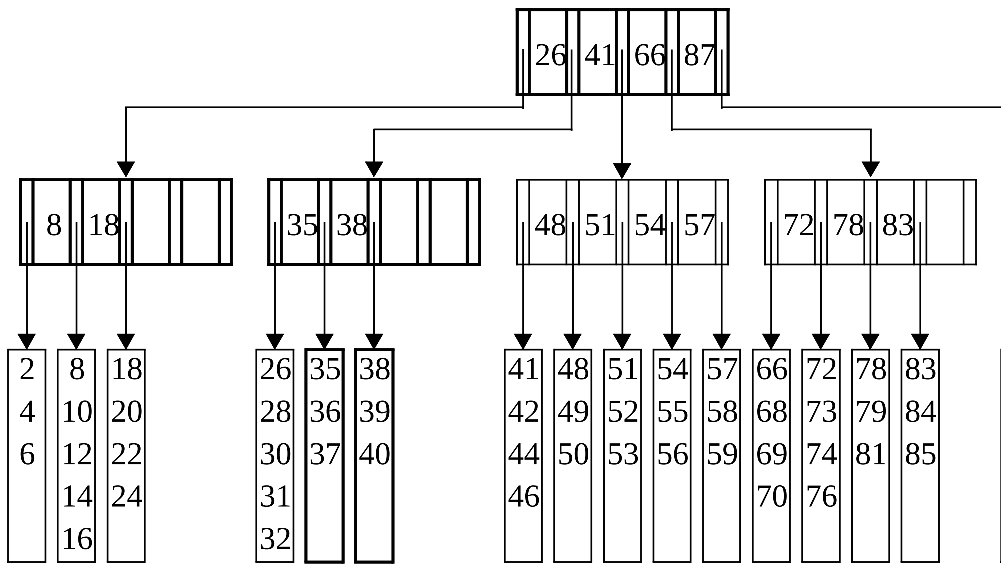
B-tree properties



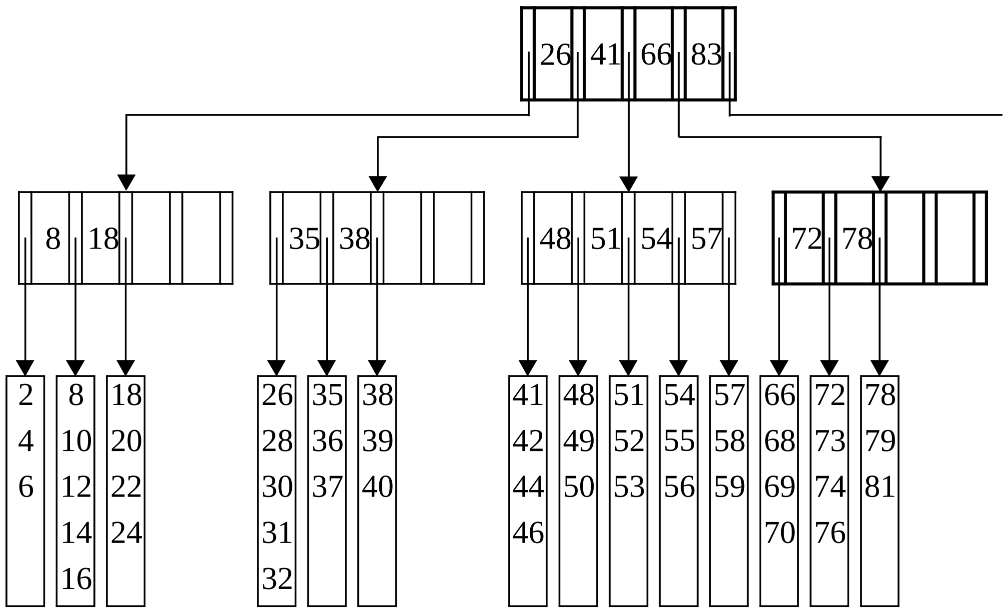
B-tree after insertion of 57 into tree in Figure 18.70



Insertion of 55 in B-tree in Figure 18.71 causes a split into two leaves



Insertion of 40 in B-tree in Figure 18.72 causes a split into two leaves and then a split of the parent node



B-tree after deletion of 99 from Figure 18.73

Chapter 19

Hash Tables


```
Hash( 89, 10 ) = 8
Hash( 18, 10 ) = 8
Hash( 49, 10 ) = 9
Hash( 58, 10 ) = 8
Hash(  9, 10 ) = 9
```

	After Insert 89	*After Insert 18*	*After Insert 49*	*After Insert 58*	*A*
0			49	49	
1				58	
2					
3					
4					
5					
6					
7					
8		18	18	18	
9	89	89	89	89	

Linear probing hash table after each insertion

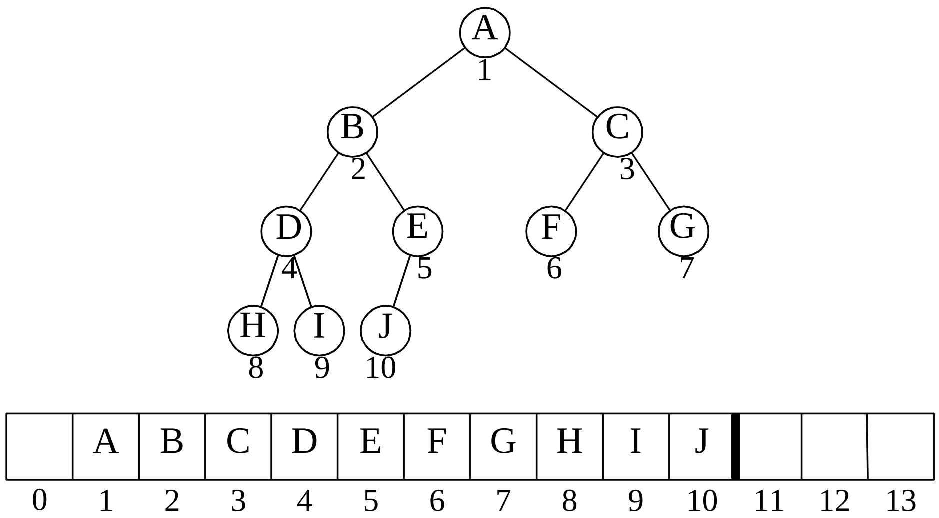

```
Hash( 89, 10 ) = 8
Hash( 18, 10 ) = 8
Hash( 49, 10 ) = 9
Hash( 58, 10 ) = 8
Hash(  9, 10 ) = 9
```

	After Insert 89	*After Insert 18*	*After Insert 49*	*After Insert 58*
0			49	49
1				
2				58
3				
4				
5				
6				
7				
8		18	18	18
9	89	89	89	89

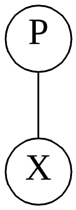
Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number)

Chapter 20

A Priority Queue: The Binary Heap

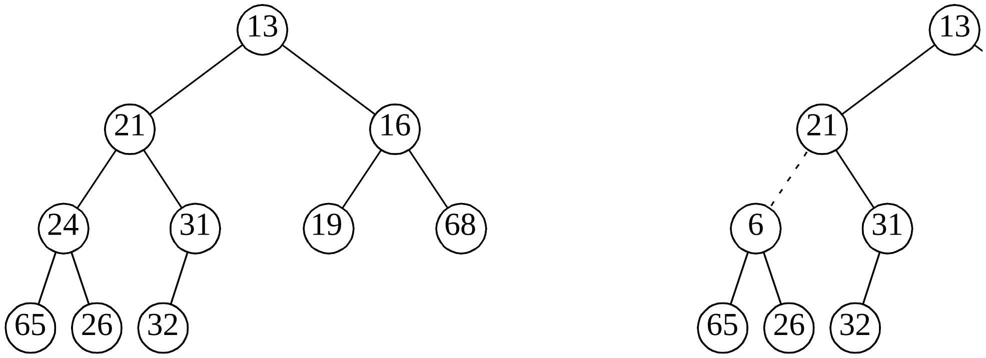


A complete binary tree and its array representation

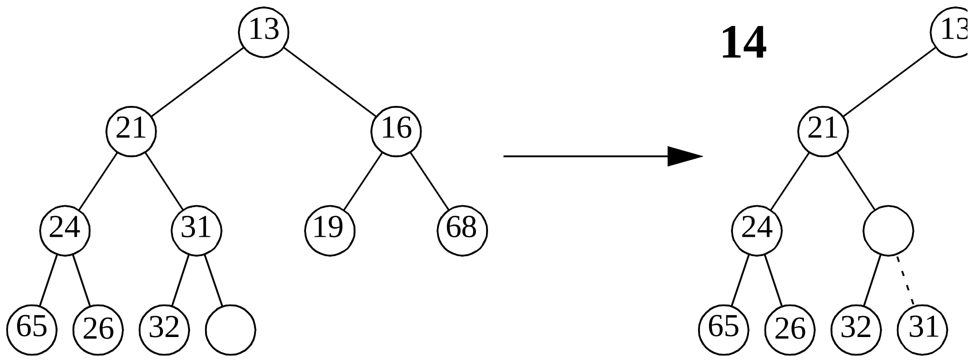


$P \leq X$

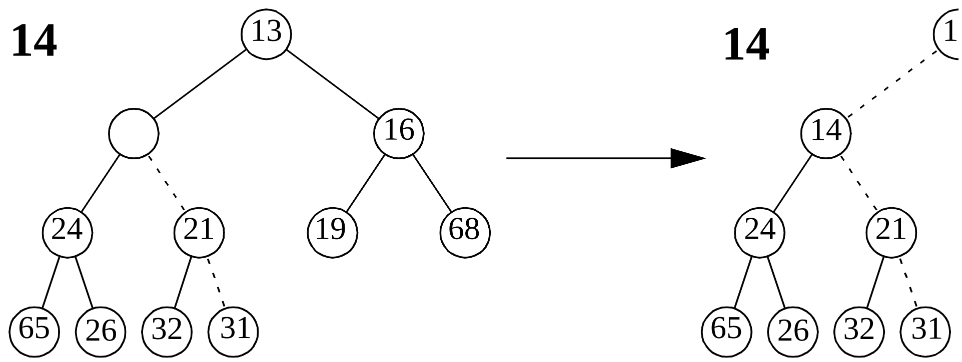
Heap order property



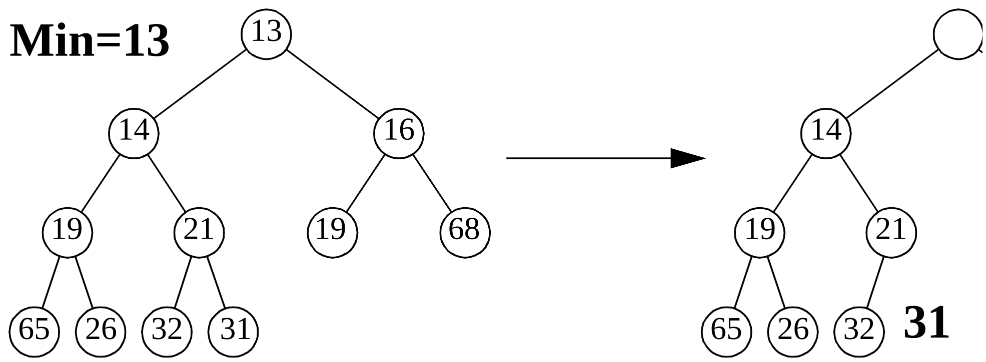
Two complete trees (only the left tree is a heap)



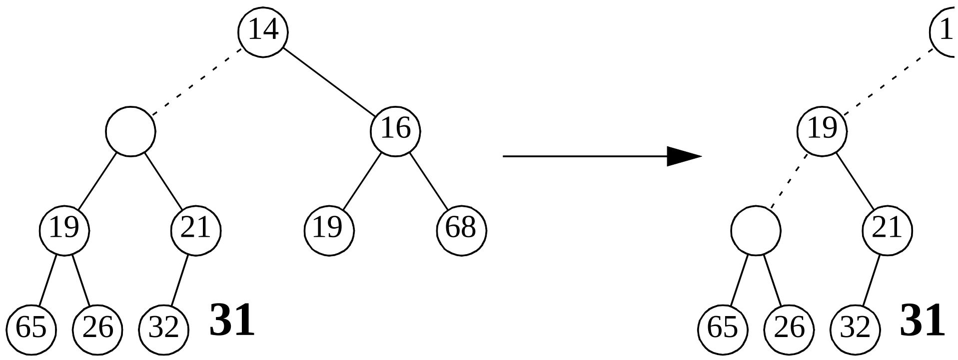
Attempt to insert 14, creating the hole and bubbling the hole up



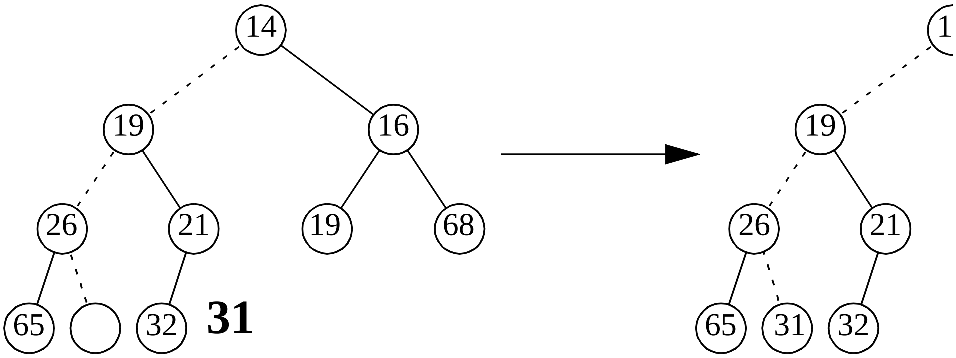
The remaining two steps to insert 14 in previous heap



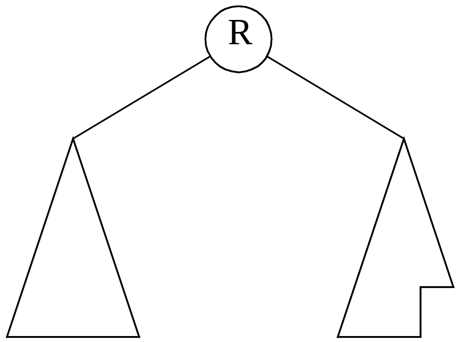
Creation of the hole at the root



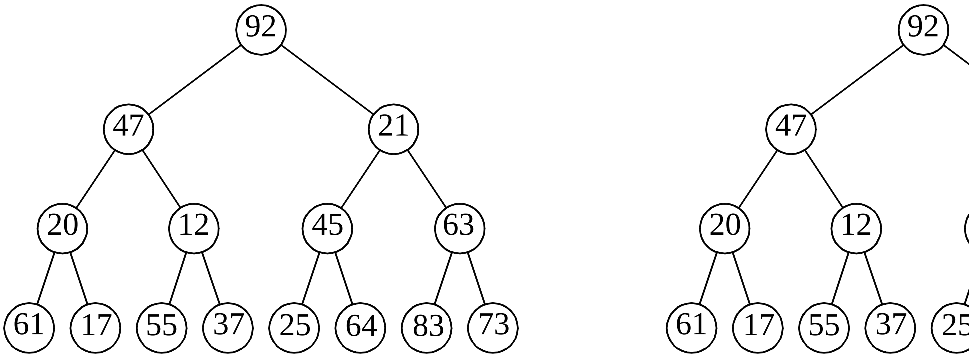
Next two steps in `DeleteMin`



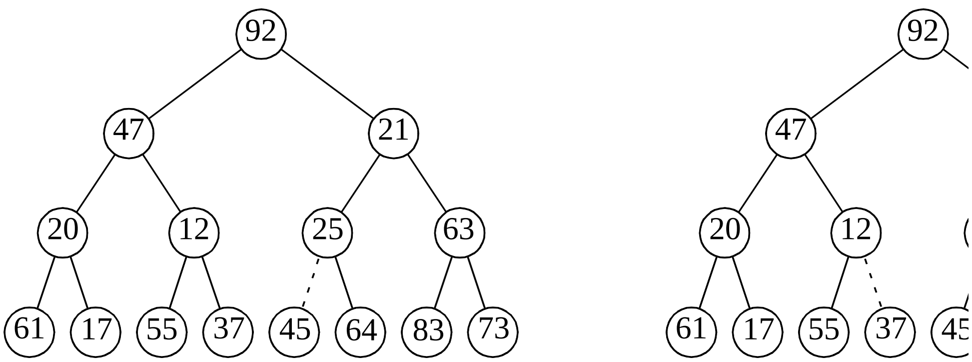
Last two steps in `DeleteMin`



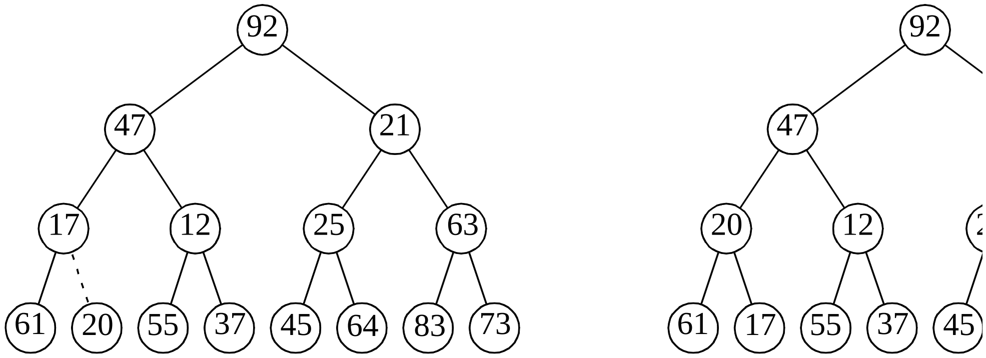
Recursive view of the heap



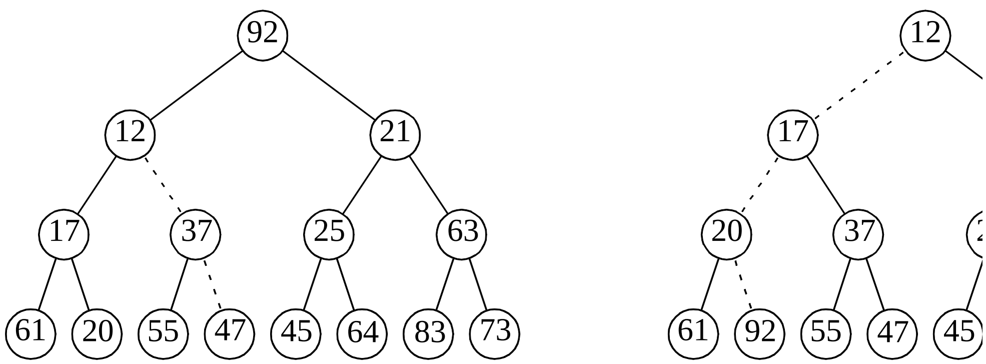
Initial heap (left); after `PercolateDown(7)` (right)



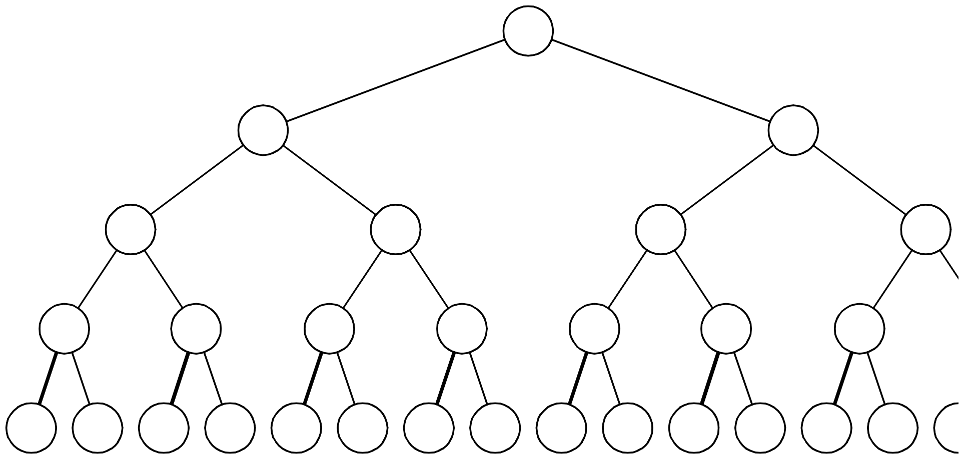
After `PercolateDown(6)` (left); after `PercolateDown(5)` (right)



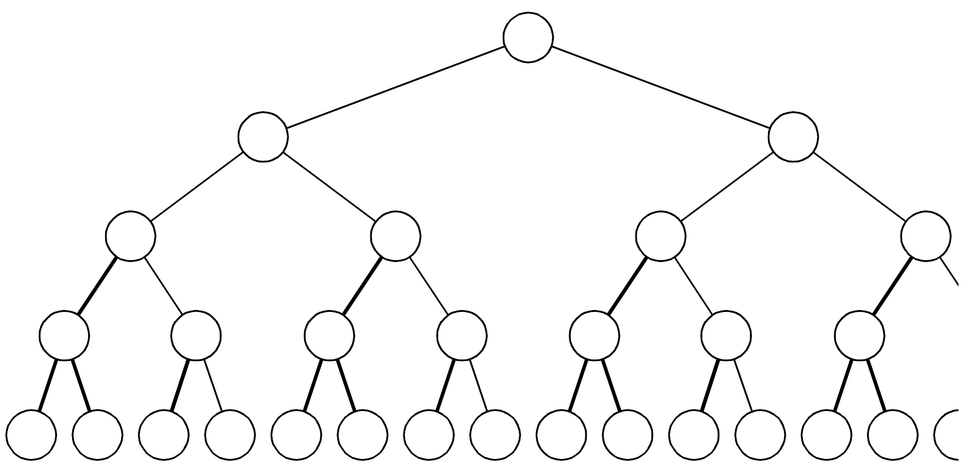
After `PercolateDown(4)` (left); after `PercolateDown(3)` (right)



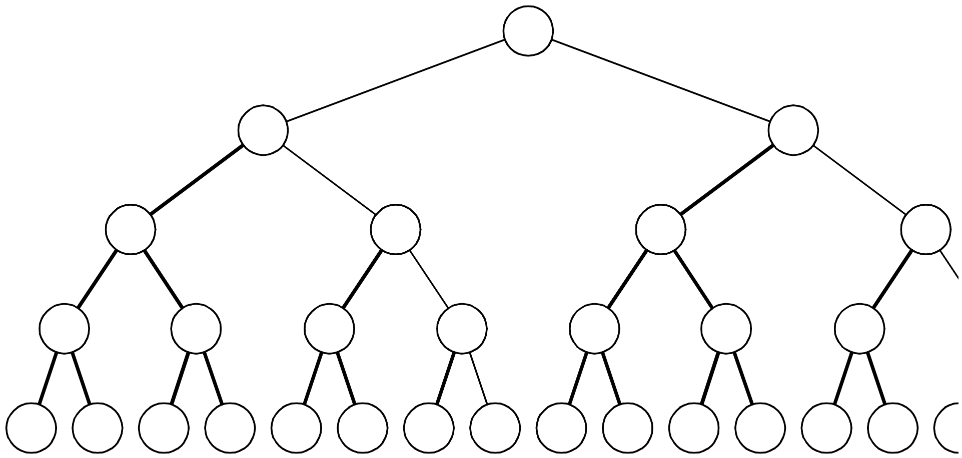
After `PercolateDown(2)` (left); after `PercolateDown(1)` and `FixHeap` terminates (right)



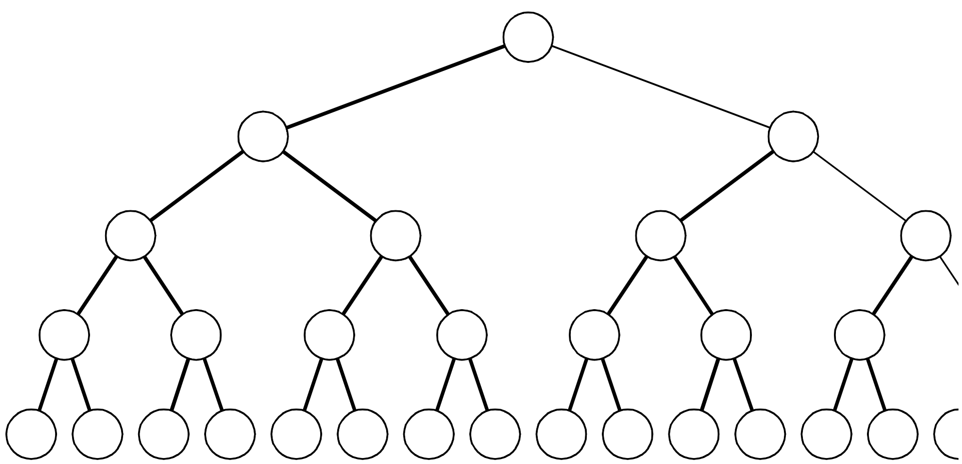
Marking of left edges for height one nodes



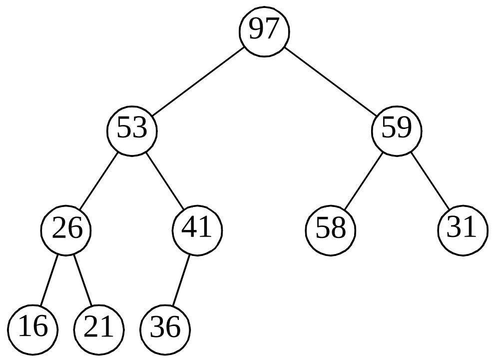
Marking of first left and subsequent right edge for height two nodes



Marking of first left and subsequent two right edges for height three nodes



Marking of first left and subsequent right edges for height 4 node

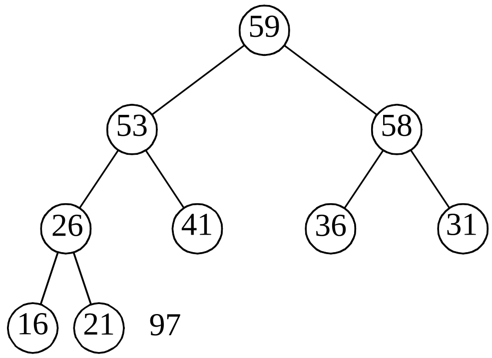


97	53	59	26	41	58	31	16	21	36				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

(Max) Heap after `FixHeap` phase

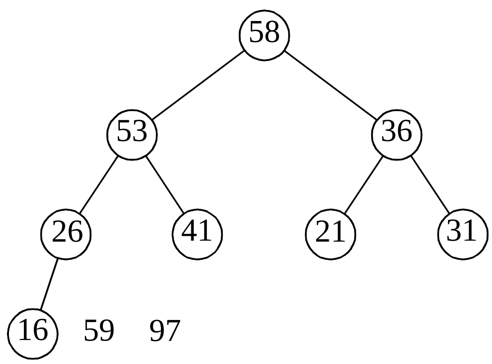
1. `Toss` each item into a binary heap.
2. Apply `FixHeap`.
3. Call `DeleteMin` *N* times; the items will exit the heap in sorted order.

Heapsort algorithm



59	53	58	26	41	36	31	16	21	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap after first `DeleteMax`



58	53	36	26	41	21	31	16	59	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap after second `DeleteMax`

A1	81	94	11	96	12	35	17	99	28	58	41	75	15
A2													
B1													
B2													

Initial tape configuration

A1							
A2							
B1	11	81	94	17	28	99	15
B2	12	35	96	41	58	75	

Distribution of length 3 runs onto two tapes

A1	11	12	35	81	94	96	15
A2	17	28	41	58	75	99	
B1							
B2							

Tapes after first round of merging (run length = 6)

A1												
A2												
B1	11	12	17	28	35	41	58	75	81	94	96	99
B2	15											

Tapes after second round of merging (run length = 12)

A1	11	12	15	17	28	35	41	58	75	81	94	96	99
A2													
B1													
B2													

Tapes after third round of merging

A1						
A2						
A3						
B1	11	81	94	41	58	75
B2	12	35	96	15		
B3	17	28	99			

Initial distribution of length 3 runs onto three tapes

A1	11	12	17	28	35	81	94	96	99
A2	15	41	58	75					
A3									
B1									
B2									
B3									

After one round of three-way merging (run length = 9)

A1													
A2													
A3													
B1	11	12	15	17	28	35	41	58	75	81	94	96	99
B2													
B3													

After two rounds of three-way merging

	Run	After						
	Const.	T3+T2	T1+T2	T1+T3	T2+T3	T1+T2	T1+T3	T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0

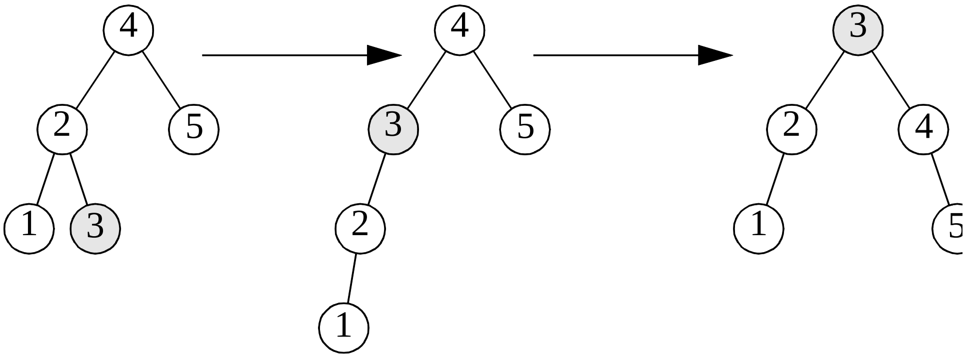
Number of runs using polyphase merge

	3 Elements in Heap Array			Output	Next Item Read
	Array[1]	Array[2]	Array[3]		
Run1	11	94	81	11	96
Run 1	81	94	96	81	12
	94	96	12	94	35
	96	35	12	96	17
	17	35	12	End of Run	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75
	58	99	75	58	End of Tape
	99		75	99	
			75	End of Run	Rebuild Heap
Run 3	75			75	

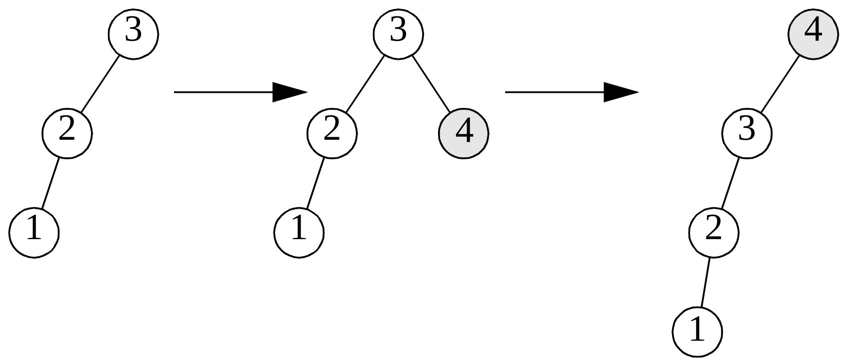
Example of run construction

Chapter 21

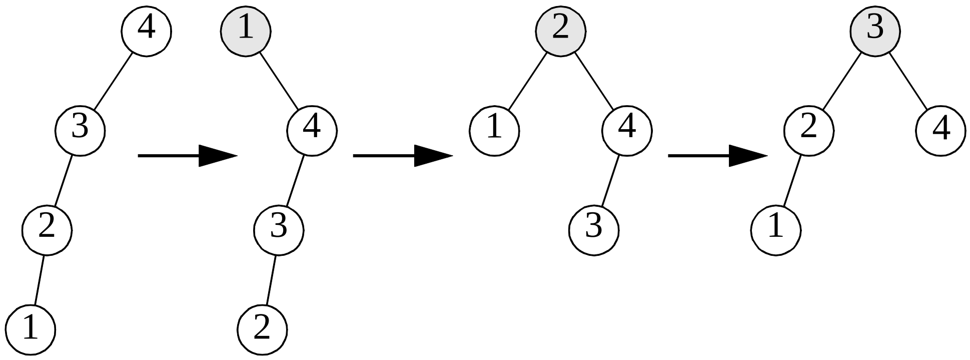
Splay Trees



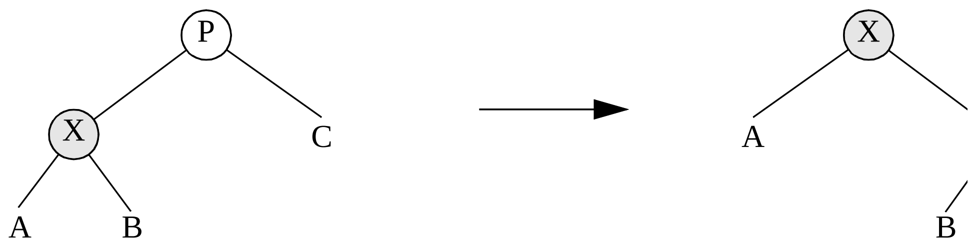
Rotate-to-root strategy applied when node 3 is accessed



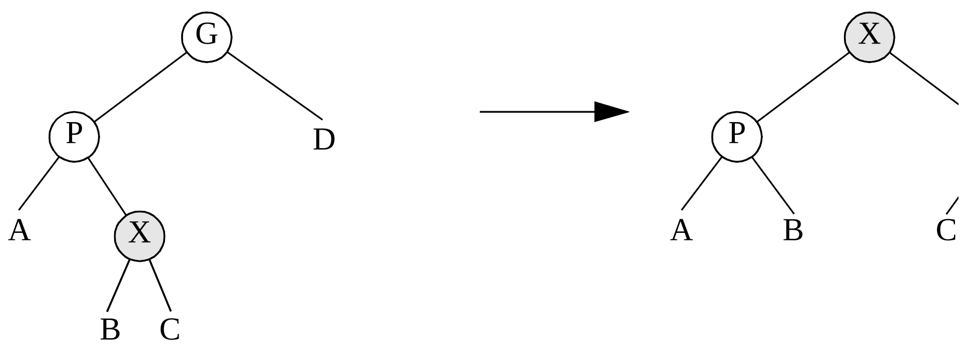
Insertion of 4 using rotate-to-root



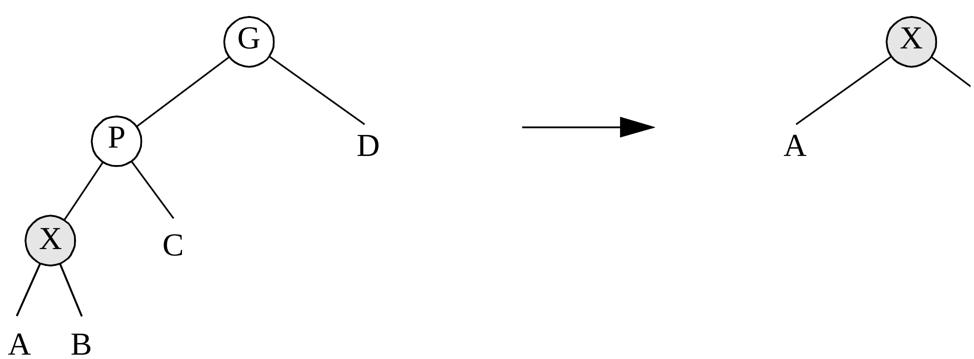
Sequential access of items takes quadratic time



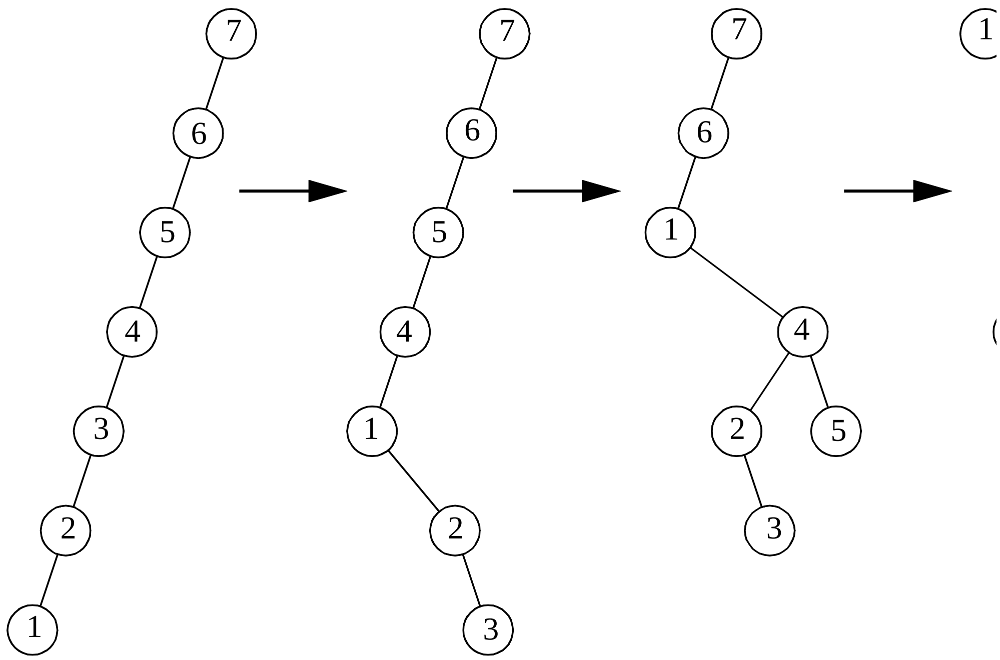
Zig case (normal single rotation)



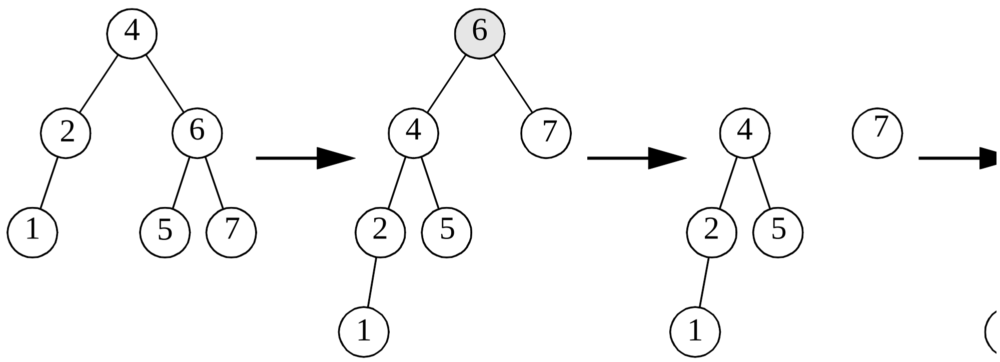
Zig-zag case (same as a double rotation); symmetric case omitted



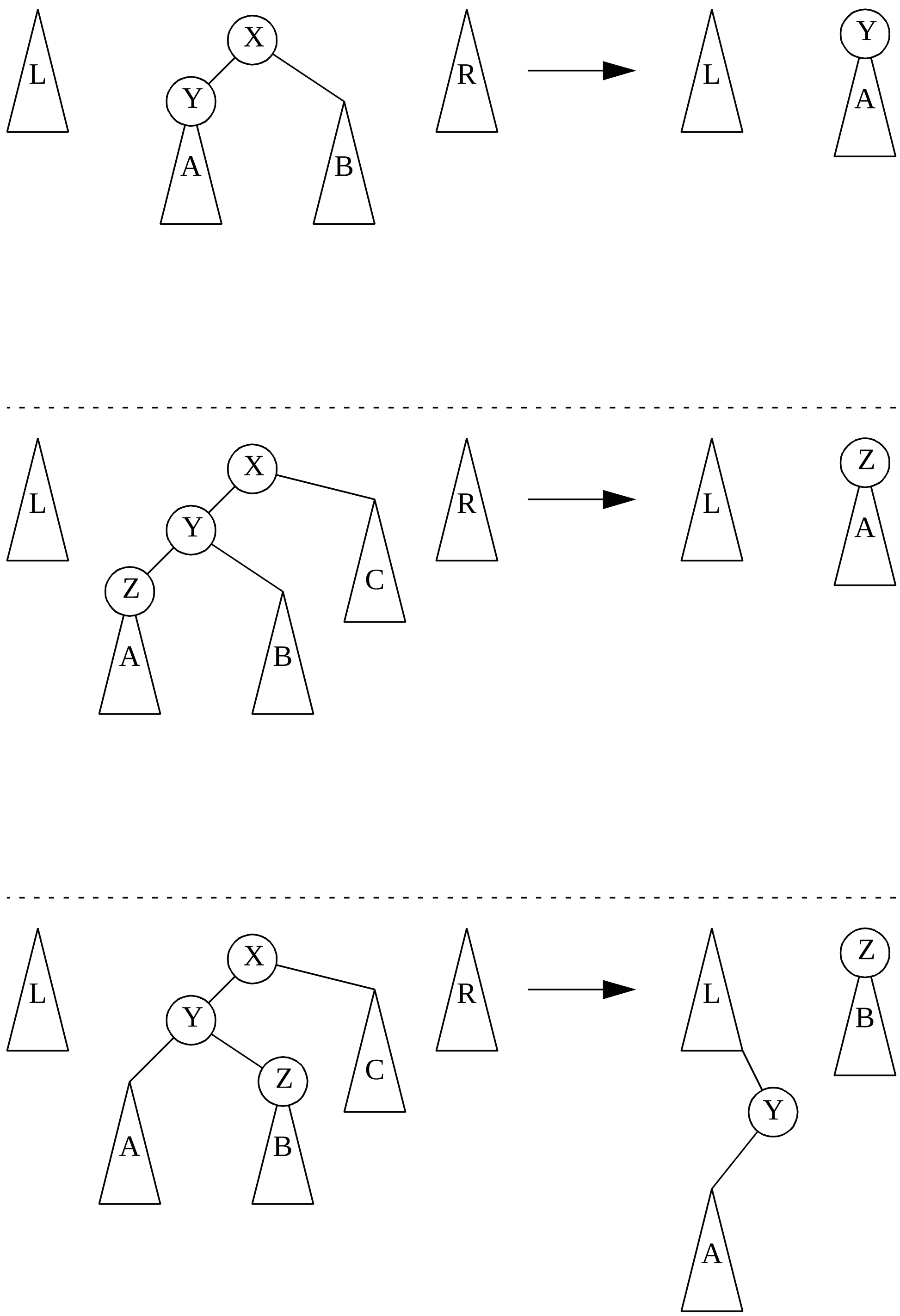
Zig-zig case (this is unique to the splay tree); symmetric case omitted



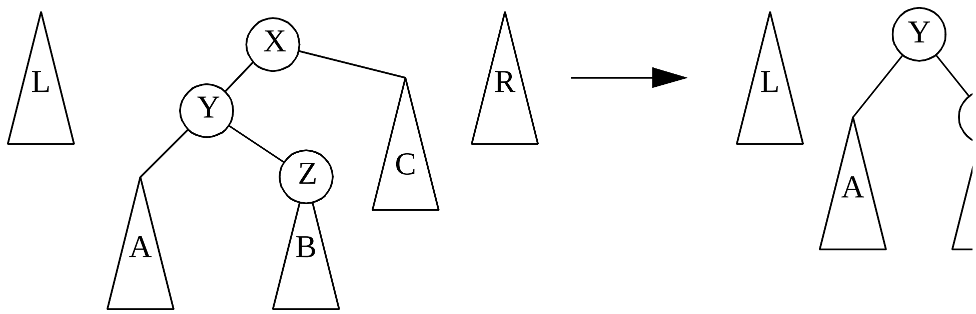
Result of splaying at node 1 (three zig-zigs and a zig)



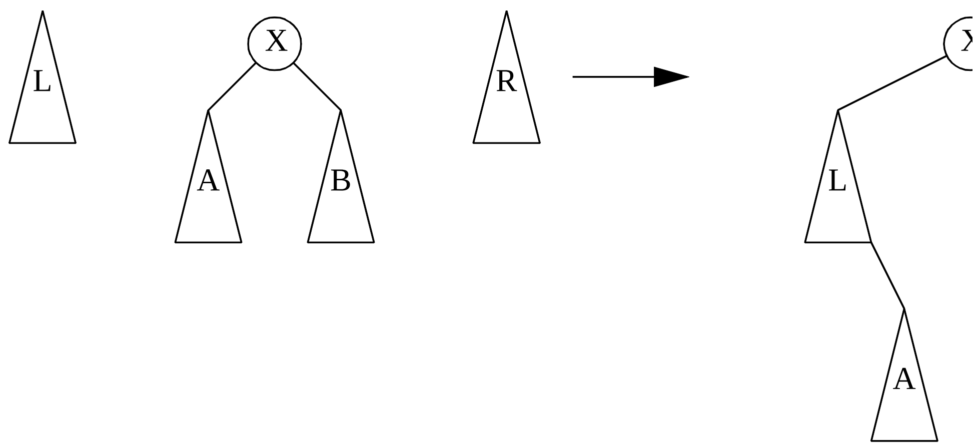
The `Remove` operation applied to node 6: First 6 is splayed to the root, leaving two subtrees; a `FindMax` on the left subtree is performed, raising 5 to the root of the left subtree; then the right subtree can be attached (not shown)



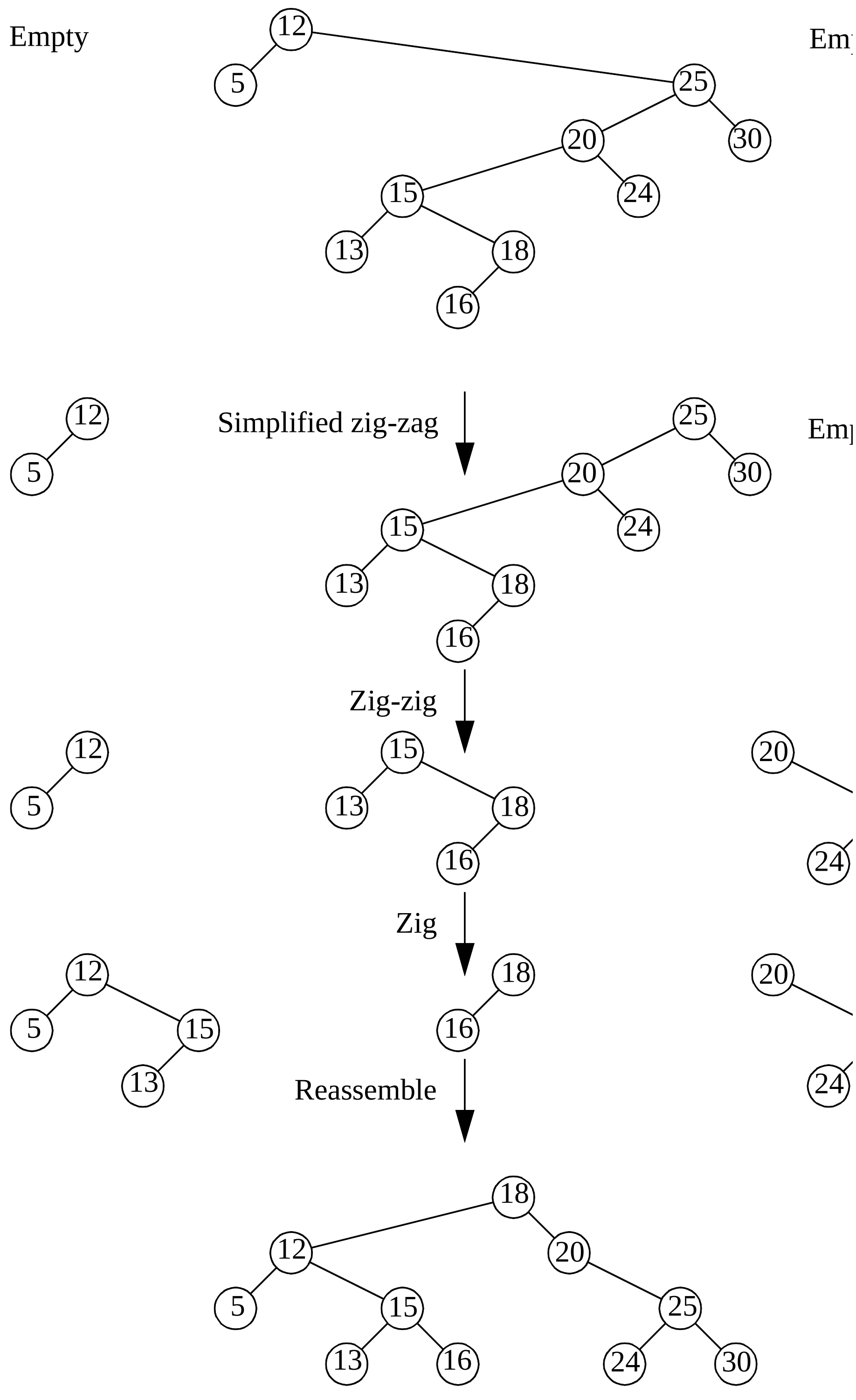
Top-down splay rotations: zig (top), zig-zig (middle), and zig-zag (bottom)



Simplified top-down zig-zag



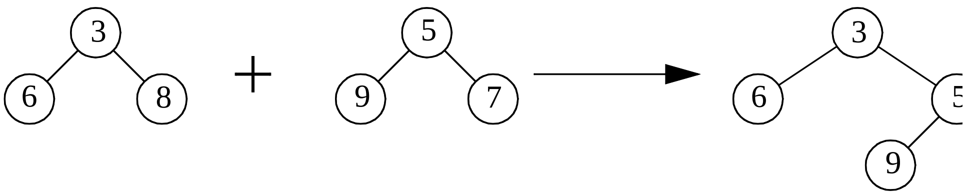
Final arrangement for top-down splaying



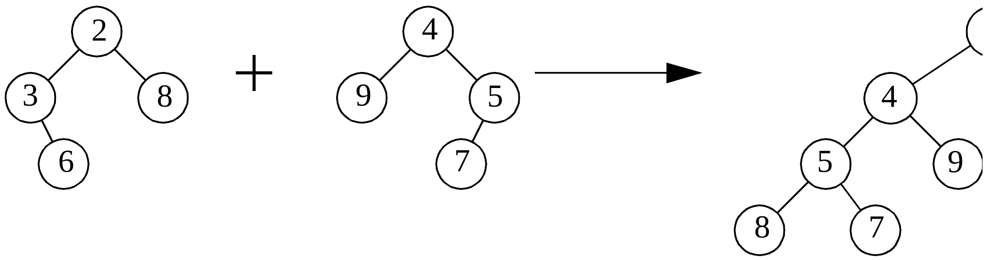
Steps in top-down splay (accessing 19 in top tree)

Chapter 22

Merging Priority Queues



Simplistic merging of heap-ordered trees; right paths are merged

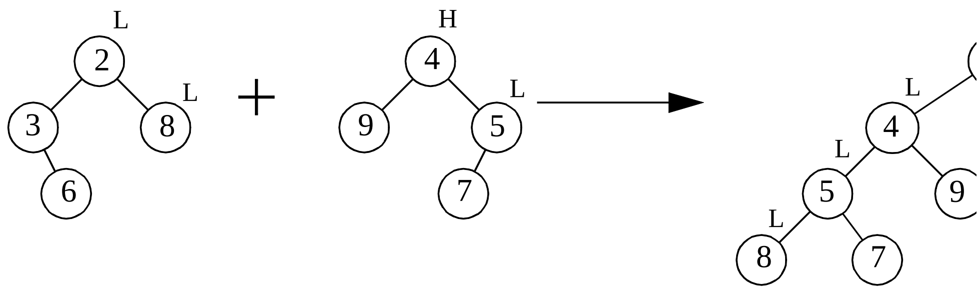


Merging of skew heap; right paths are merged, and the result is made a left path

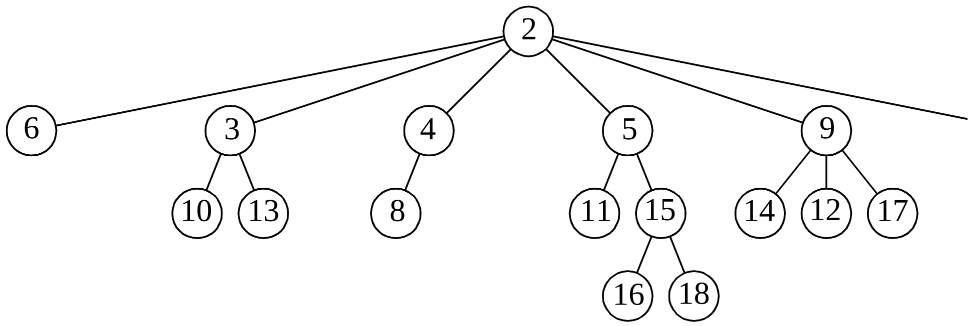
A recursive viewpoint is as follows: Let *S* be the tree with the smaller root, and let *R* be the other tree.

1. If one tree is empty, the other can be used as the merged result.
2. Otherwise, let *Temp* be the right subtree of *L*.
3. Make *L*'s left subtree its new right subtree.
4. Make the result of the recursive merge of *Temp* and *R* the new left subtree of *L*.

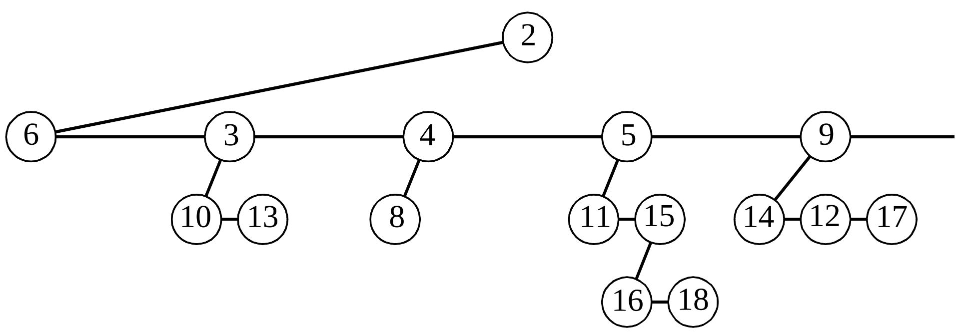
Skew heap algorithm (recursive viewpoint)



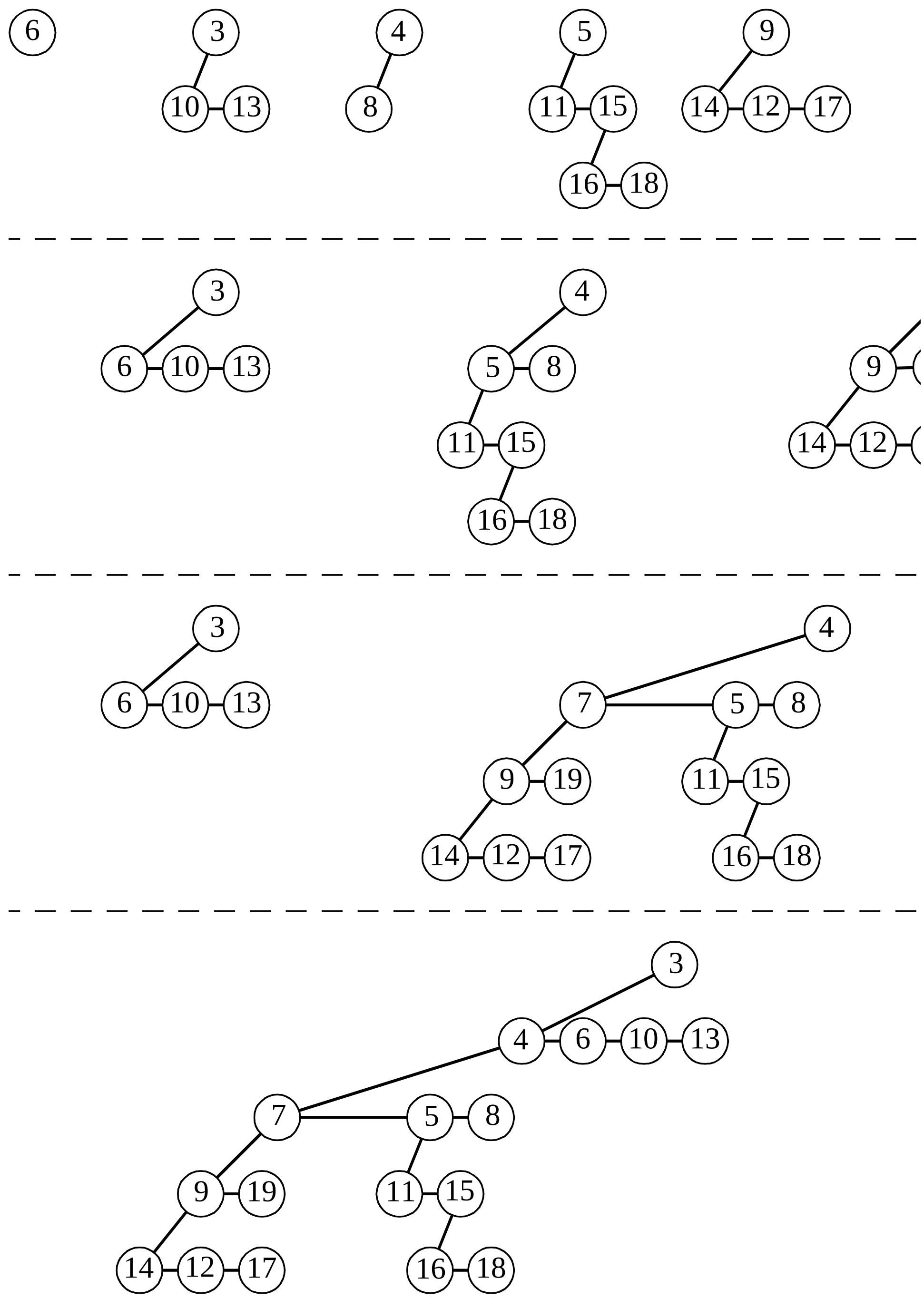
Change in heavy/light status after a merge



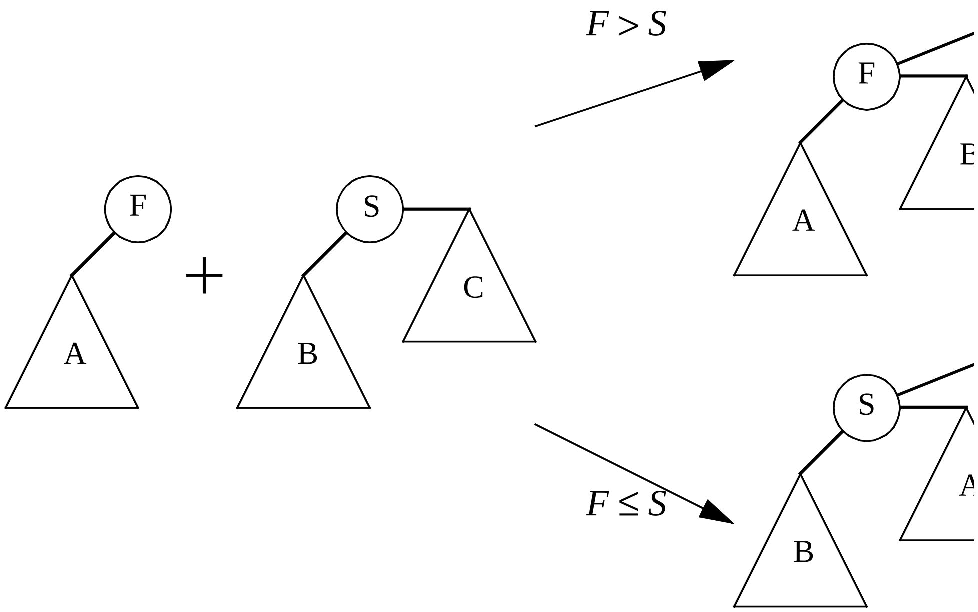
Abstract representation of sample pairing heap



Actual representation of above pairing heap; dark line represents a pair of pointers that connect nodes in both directions



Recombination of siblings after a `DeleteMin`; in each merge the larger root tree is made the left child of the smaller root tree: (a) the resulting trees; (b) after the first pass; (c) after the first merge of the second pass; (d) after the second merge of the second pass



`CompareAndLink` merges two trees

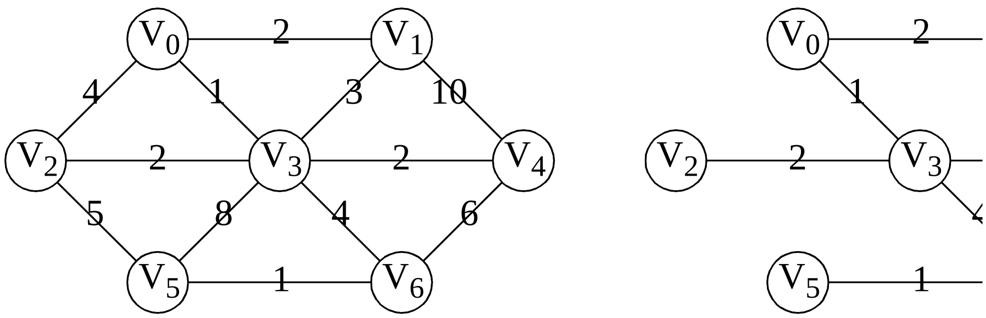
Chapter 23

The Disjoint Set Class

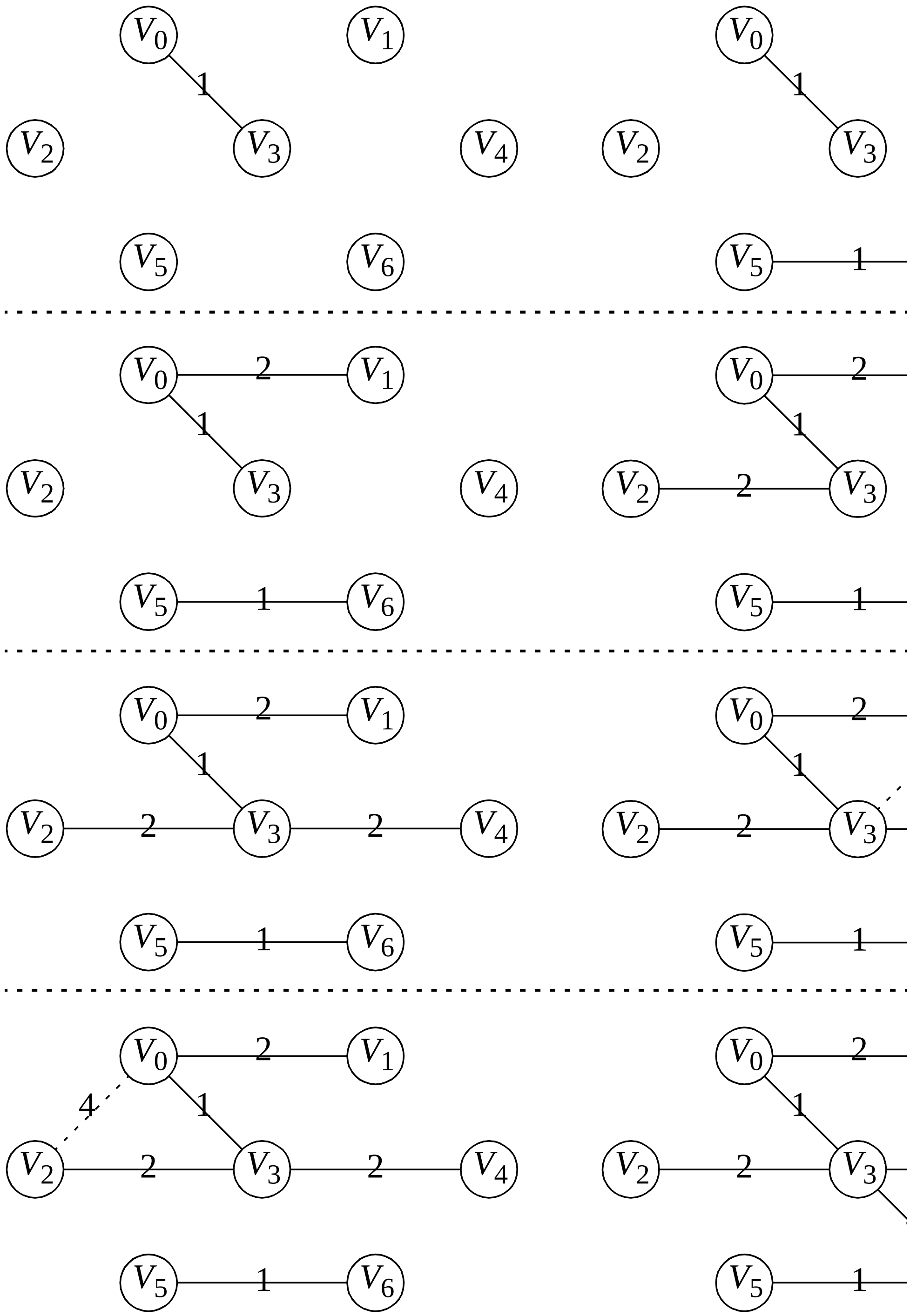
A *relation* R is defined on a set S if for every pair of elements ( ), $a, b \in S$, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, then we say that a is related to b. An *equivalence relation* is a relation R that satisfies three properties:

- *Reflexive*: $a\ R\ a$ is true for all $a \in S$
- *Symmetric*: $a\ R\ b$ if and only if $b\ R\ a$
- *Transitive:* $a\ R\ b$ and $b\ R\ c$ implies that $a\ R\ c$

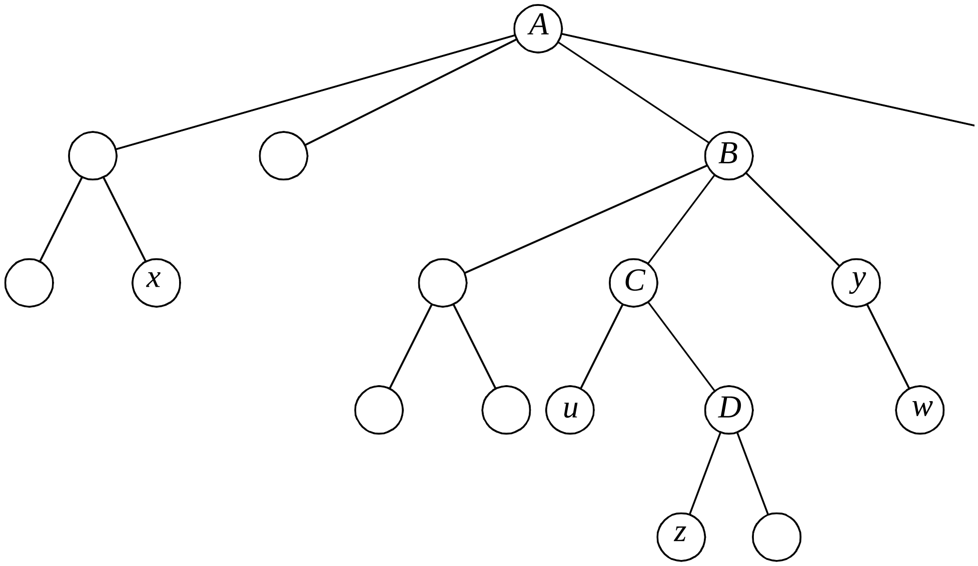
Definition of equivalence relation



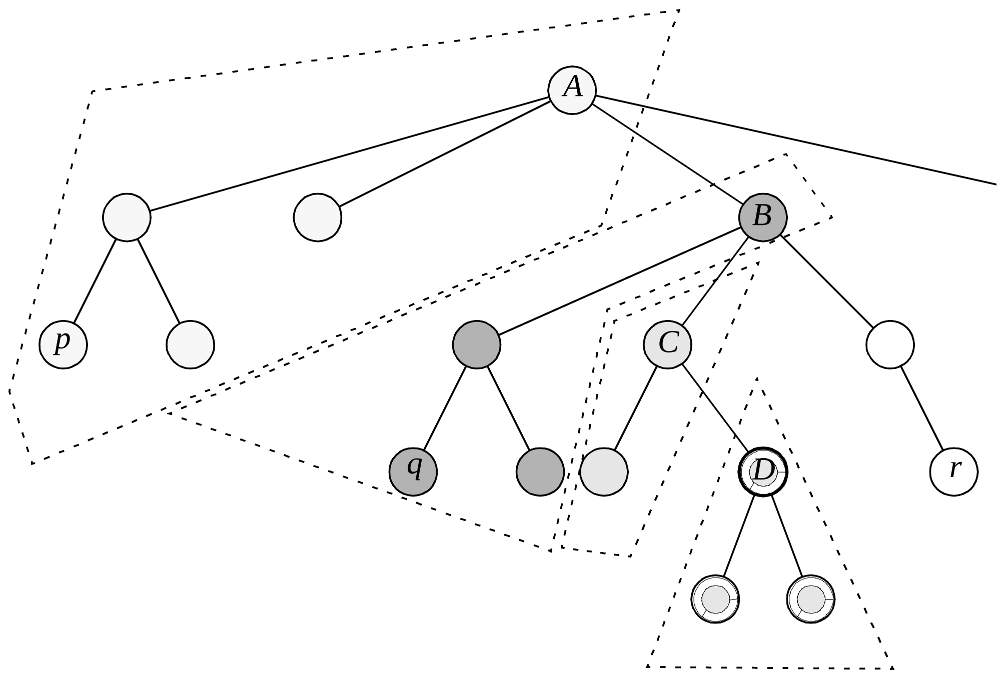
A graph *G* (left) and its minimum spanning tree



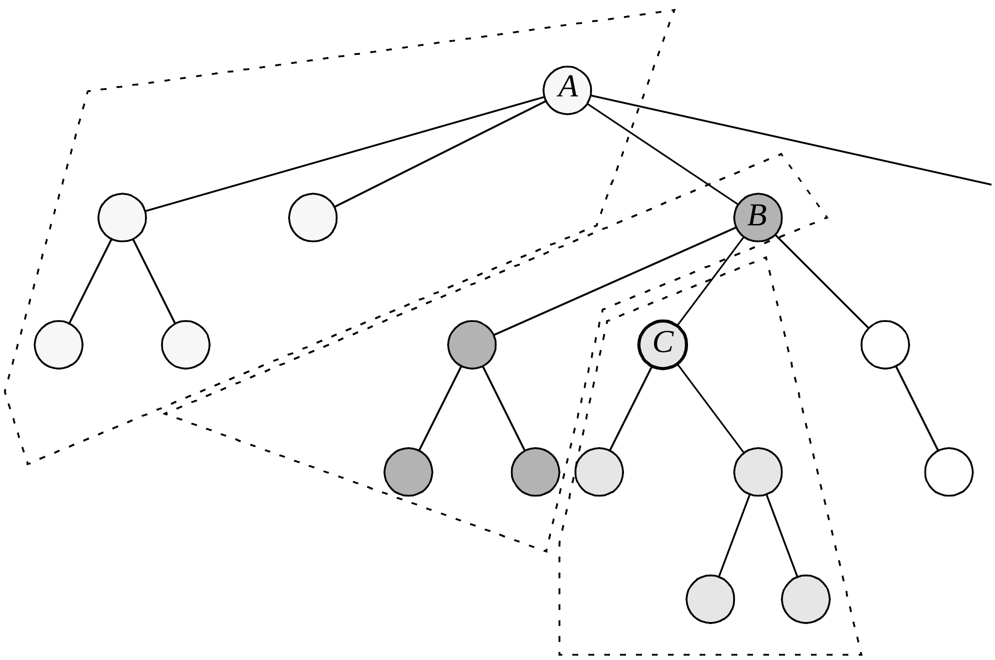
Kruskal's algorithm after each edge is considered



The nearest common ancestor for each request in the pair sequence (*x*,*y*), (*u*,*z*), (*w*,*x*), (*z*,*w*), (*w*,*y*), is *A*, *C*, *A*, *B*, and *y*, respectively

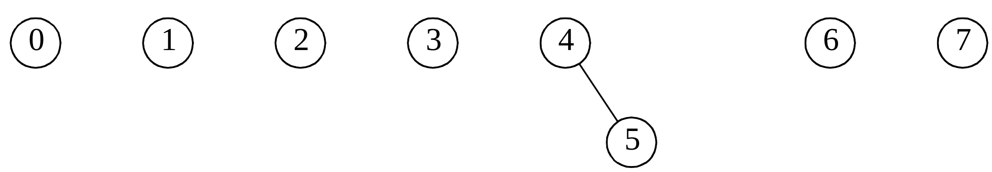


The sets immediately prior to the return from the recursive call to *D*; *D* is marked as visited and *NCA*(*D*, *v*) is *v*'s anchor to the current path

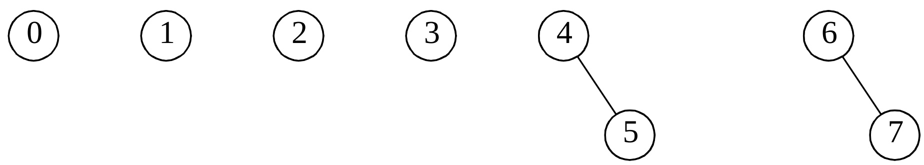


After the recursive call from *D* returns, we merge the set anchored by *D* into the set anchored by *C* and then compute all *NCA*(*C*, *v*) for nodes *v* that are marked prior to completing *C*'s recursive call

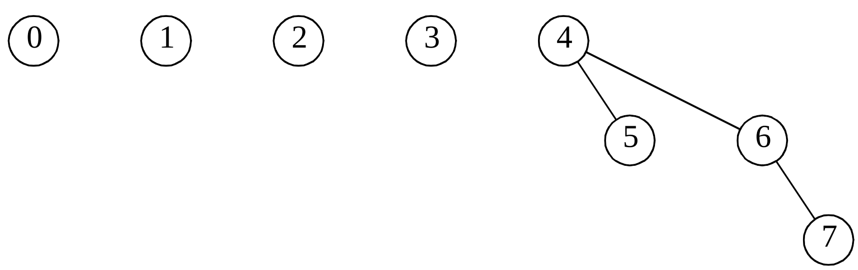
Forest and its eight elements, initially in different sets



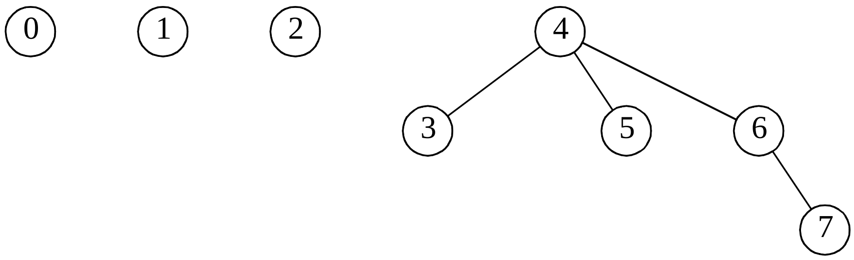
Forest after `Union` of trees with roots 4 and 5



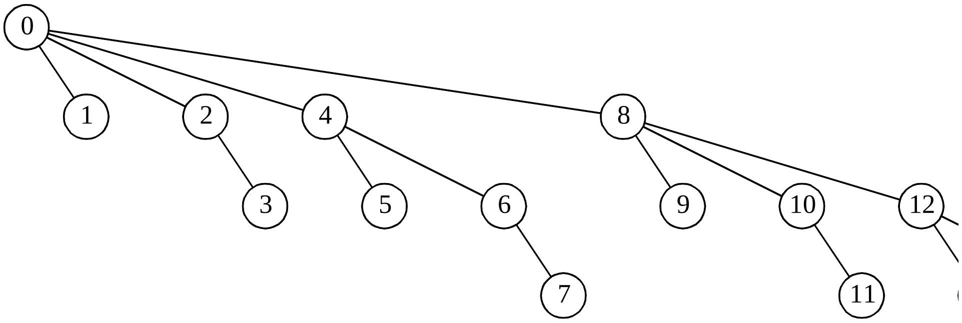
Forest after `Union` of trees with roots 6 and 7



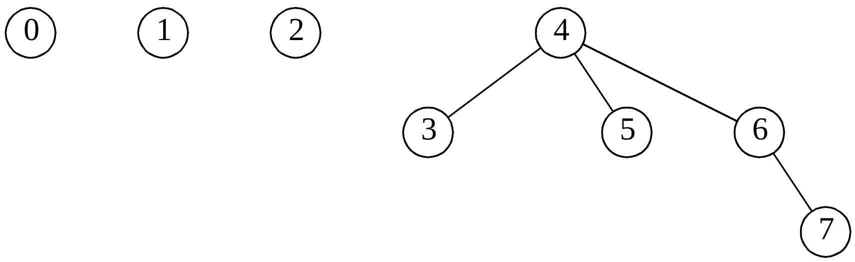
Forest after `Union` of trees with roots 4 and 6



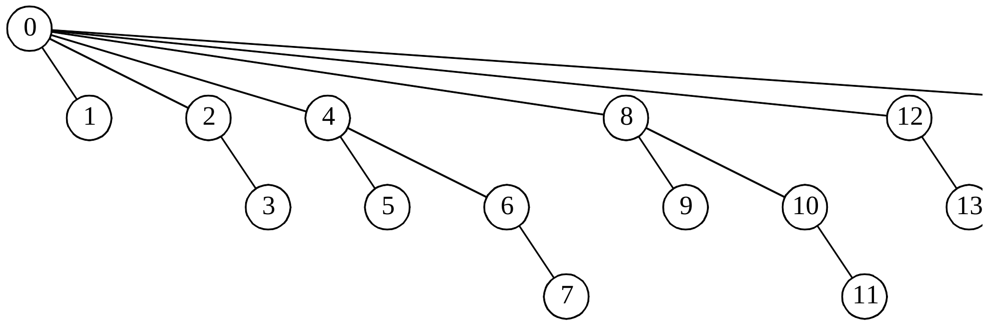
Forest formed by union-by-size, with size encoded as a negative number



Worst-case tree for N=16



Forest formed by union-by-height, with height encoded as a negative number



Path compression resulting from a `Find`(14) on the tree in Figure 23.12

Ackerman’s function is defined as:

$$A(\quad) = 2^j \qquad j \ge 1$$

$$A(\quad) = A(\quad) \qquad i \ge 2$$

$$A(\quad) = A(\quad) \qquad i, j \ge 2$$

From this, we define the inverse Ackerman’s function as

$$\alpha(\quad) = min\{ \quad | \quad \lfloor \quad \rfloor \quad \}$$

Ackerman’s function and its inverse

To incorporate path compression into the proof, we use the following fancy accounting: For each node *v* on the path from the accessed node *i* to the root, we deposit one penny under one of two accounts:

1. If *v* is the root, or if the parent of *v* is the root, or if the parent of *v* is in a different rank group from *v*, then charge one unit under this rule. This deposits an American penny into the kitty.
2. Otherwise, deposit a Canadian penny into the node.

Accounting used in union-find proof

Group	Rank
0	0
1	1
2	2
3	3,4
4	5 through 16
5	17 through 65536
6	65537 through 2^{65536}
7	Truly huge ranks

Actual partitioning of ranks into groups used in the union-find proof

List of Transparencies

Chapter 20 A Priority Queue: The Binary Heap 221

Chapter 21 Splay Trees 243