COMP2230/6230 Algorithms

Lecture 4

Professor Ljiljana Brankovic

Lecture Overview

- Data Structures

Data Structures - Revision

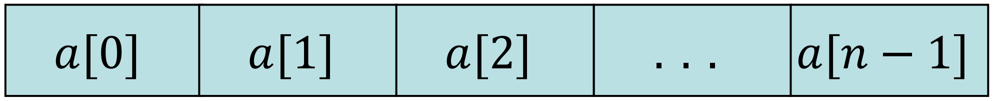
- Most algorithms operate on *data*.
- Typically, an algorithm takes data as input, manipulates them, and produces data as output.
- *Data structure* is a structure (organisation, arrangement) of data, or, more specifically, related data items used by an algorithm.
- An *abstract data type (ADT)* comprises both *data* and *functions* that operate on data. We shall define some abstract data types by listing the functions that operate on them.
- "*Abstract*" means that an implementation is and not specified.

Some Fundamental Data Structures

- Linear data structures:
  - Arrays
  - Linked Lists
  - Stacks
  - Queues
- Graphs – general
- Trees – general
- Binary Trees
- Heaps
- Sets (Disjoint Sets)

Arrays

- A one-dimensional array is a sequence of n items, each having an index between 0 and $n-1$ associated with it.
- All data items are of the same data type (e.g., integers).
- In an array, an element (data item) can be accessed directly by specifying its index; thus, arrays provide constant time access to elements.
- However, inserting or deleting an element in an array takes $\Theta(n)$.



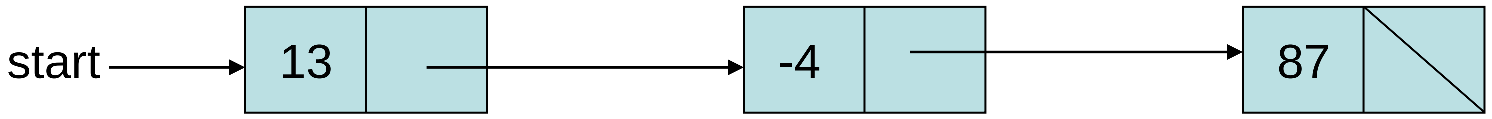
Linked Lists

- Unlike an array, a linked list provides a constant time insertion and deletion anywhere in the list, but in the worst case it takes $\Theta(n)$ time to access an element.

- A linked list can be implemented as a list of nodes, where each node has a field ***data*** and a field ***next***, which references the next node in the list.

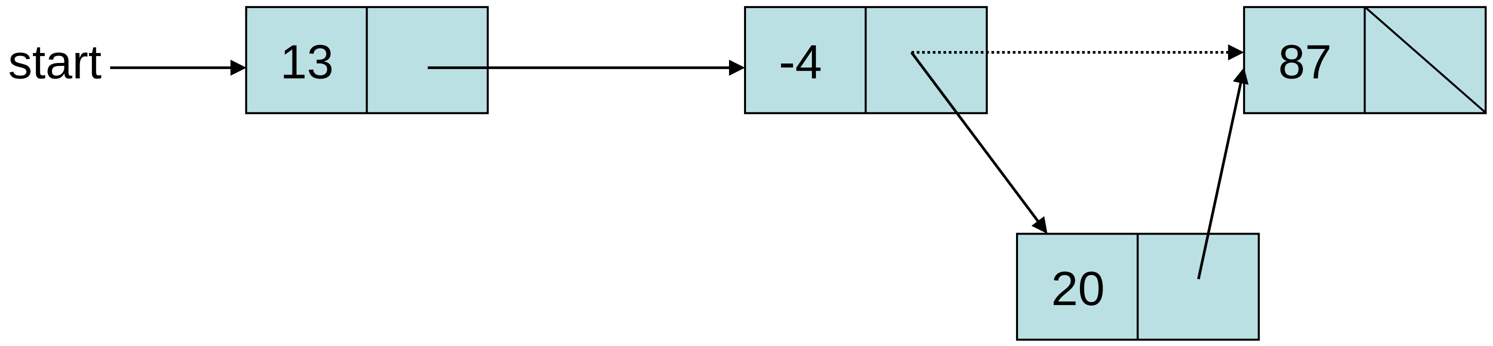
- A list also contains a variable start that references the first node in the list.

- The ***next*** field in the last node in **null**.



Linked Lists

- Inserting in a linked list :



Algorithm 3.3.1 Inserting in a Linked List

This algorithm inserts the value *val* after the node referenced by *pos*. The operator, new, is used to obtain a new *node*.

```
Input Parameters: val,pos
Output Parameters: None
insert(val,pos) {
    temp = new node
    temp.data = val
    temp.next = pos.next
    pos.next = temp
}
```

Algorithm 3.3.2 Deleting in a Linked List

This algorithm deletes the node after the node referenced by *pos*.

```
Input Parameters: pos
Output Parameters: None
delete(pos) {
    pos.next = pos.next.next
}
```

Algorithm 3.3.3 Printing the Data in a Linked List

This algorithm prints the data in each node in a linked list. The first node is referenced by *start*.

```
Input Parameters: start
Output Parameters: None
print(start) {
    while (start != null) {
        println(start.data)
        start = start.next
    }
}
```

Stacks

A stack is an abstract data type (ADT) that has the following functions:

- *stack_init()*: initialise stack (make it empty)
- *empty()*: return true if the stack is empty and return false if the stack is not empty
- *push(val)*: add the item *val* to the stack
- *pop()*: remove the item most recently added to the stack
- *top()*: return the item most recently added to the stack but do not remove it

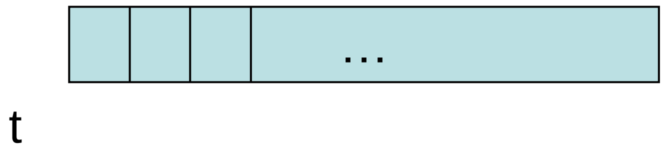
Implementing stacks using arrays

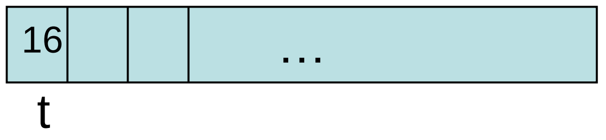
- Note that stack is a Last In First Out structure (LIFO).
- Variable `t` is used to denote the top of the stack.
- When an item is pushed onto the stack, `t` is incremented and the item is put in the cell at index `t`.
- When an item is *popped* off the stack, `t` is decremented.

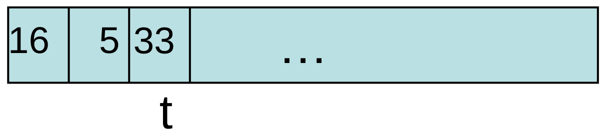
Example 1

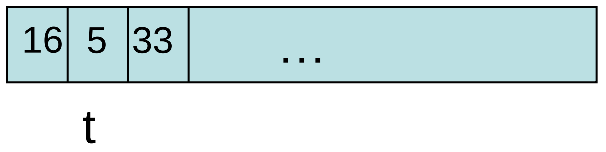
Starting from empty stack:

```
s.push(16)
s.push(5)
s.push(33)
s.pop()
```









Algorithm 3.2.2 Initializing a Stack

This algorithm initializes a stack to empty. An empty stack has $t = -1$.

```
Input Parameters: None
Output Parameters: None
stack_init() {
    t = -1
}
```

Algorithm 3.2.3 Testing for an Empty Stack

This algorithm returns true if the stack is empty or false if the stack is not empty. An empty stack has $t = -1$.

```
Input Parameters: None
Output Parameters: None
empty() {
    return $t$ == -1
}
```

Algorithm 3.2.4 Adding an Element to a Stack

This algorithm adds the value *val* to a stack. The stack is represented using an array *data*. The algorithm assumes that the array is not full. The most recently added item is at index t unless the stack is empty, in which case, $t = -1$.

```
Input Parameters: val
Output Parameters: None
push(val) {
    $t$ = $t$ + 1
    data[$t$] = val
}
```

Algorithm 3.2.5 Removing an Element From a Stack

This algorithm removes the most recently added item from a stack. The algorithm assumes that the stack is not empty. The most recently added item is at index t.

```
Input Parameters: None
Output Parameters: None
pop() {
    t = t - 1
}
```

Algorithm 3.2.6 Returning the Top Element in a Stack

This algorithm returns, but does not remove, the most recently added item in a stack. The algorithm assumes that the stack is not empty. The stack is represented using an array *data*. The most recently added item is at index *t*.

```
Input Parameters: None
Output Parameters: None
top() {
    return data[t]
}
```

Example 2: Algorithm 3.2.7 One

This algorithm returns true if there is exactly one element on the stack. The code uses the abstract data type stack.

```
Input Parameter: s (the stack)
Output Parameters: None
one(s) {
    if (s.empty())
        return false
    val = s.top()
    s.pop()
    flag = s.empty()
    s.push(val)
    return flag
}
```

Queues

A queue is very similar to a stack; the only difference is that when an item is deleted from a queue, then the least recently added item is deleted, not the most recently added as in a stack – First In First Out - FIFO.

Queues

The functions on a queue are:

- *queue_init():* initialise the queue (make it empty)
- *empty():* return true if the queue is empty and return false if the queue is not empty
- *enqueue(val):* add the item *val* to the queue
- *dequeue():* remove the item <u>least</u> recently added to the queue
- *front():* return the item least recently added to the queue but do not remove it

Queues

Queue can also be implemented using arrays.

- *Variables r and f are used to denote the rear and the front of a queue.*

- *For an empty queue, $r = f = -1$.*

- *When an item is added to an empty queue, both r and f are set to 0 and the item is put in the cell at index 0.*

- *When an item is added to a non-empty queue, r is incremented and the item is put in the cell at index r. If r is the index of the last cell in the array, it is set to 0.*

Queues

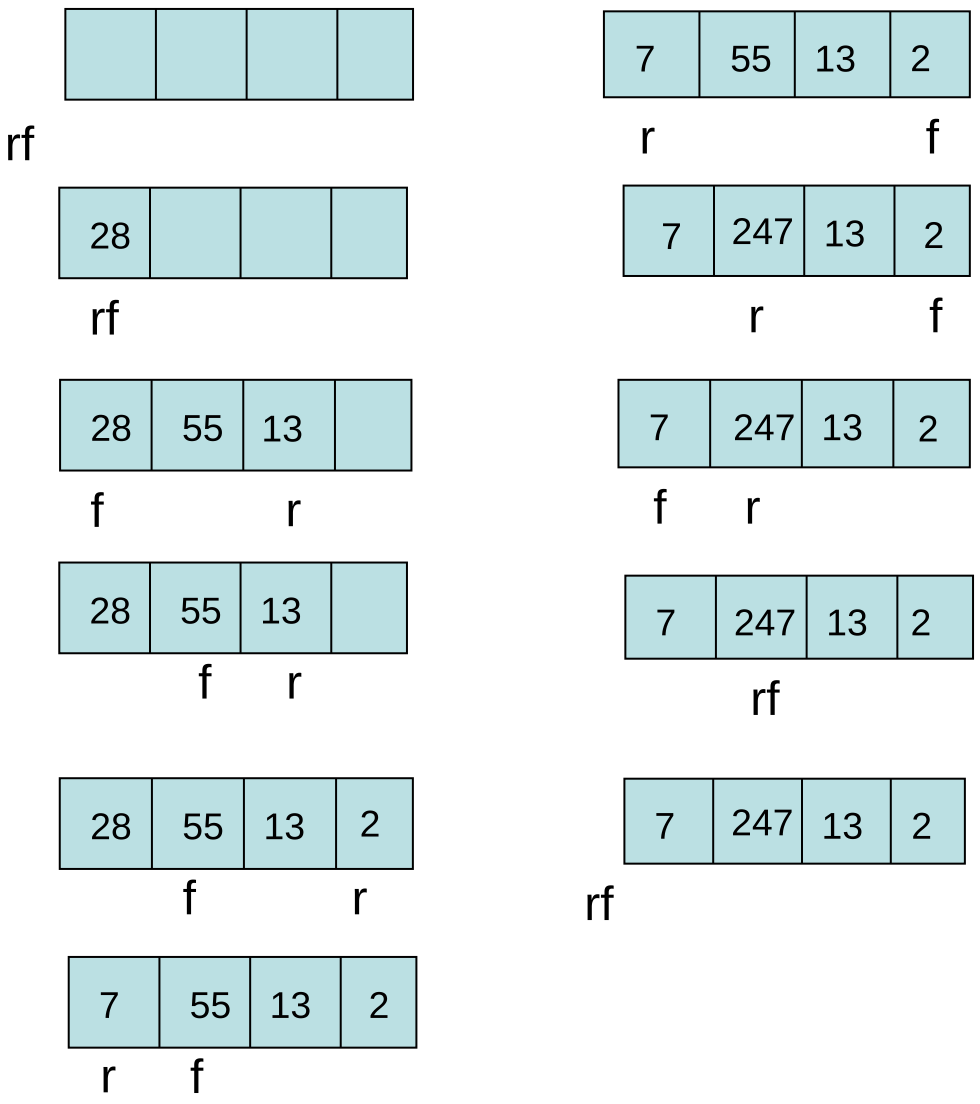
- When an item is deleted from the front of a queue, f gets incremented. If f is the index of the last cell in the array, it is set to 0.

- When an item is deleted from a queue and queue becomes empty, both r and f are set to -1.

Example 3

Starting from an empty queue:

```
1. q.enqueue(28)
2. q.enqueue(55)
3. q.enqueue(13)
4. q.dequeue()
5. q.enqueue(2)
6. q.enqueue(7)
7. q.dequeue()
8. q.dequeue()
9. q.enqueue(247)
10.q.dequeue()
11.q.dequeue()
12.q.dequeue()
```



Algorithm 3.2.10 Initializing a Queue

This algorithm initializes a queue to empty. An empty queue has $r = f = -1$.

```
Input Parameters: None
Output Parameters: None
queue_init() {
    $r$ = $f$ = - 1
}
```

Algorithm 3.2.11 Testing for an Empty Queue

This algorithm returns true if the queue is empty or false if the queue is not empty. An empty queue has $r = f = -1$.

```
Input Parameters: None
Output Parameters: None
empty() {
    return $r$ == - 1
}
```

Algorithm 3.2.12 Adding an Element to a Queue

This algorithm adds the value *val* to a queue. The queue is represented using an array *data* of size *SIZE*. The algorithm assumes that the queue is not full. The most recently added item is at index r (rear), and the least recently added item is at index f (front). If the queue is empty, $r = f = -1$.

```
Input Parameters: val
Output Parameters: None
enqueue(val) {
    if (empty())
        r = f = 0
    else {
        r = r + 1
        if (r == SIZE)
            r = 0
    }
    data[r] = val
}
```

Algorithm 3.2.13 Removing an Element From a Queue

This algorithm removes the least recently added item from a queue. The queue is represented using an array of size $SIZE$. The algorithm assumes that the queue is not empty. The most recently added item is at index r (rear), and the least recently added item is at index f (front). If the queue is empty, $r = f = -1$.

```
Input Parameters: None
Output Parameters: None
dequeue() {
    // does queue contain one item?
    if ($r$ == $f$)
        $r$ = $f$ = -1
    else {
        $f$ = $f$ + 1
        if ($f$ == $SIZE$)
            $f$ = 0
    }
}
```

Algorithm 3.2.14 Returning the Front Element in a Queue

This algorithm returns, but does not remove, the least recently added item in a queue. The algorithm assumes that the queue is not empty. The queue is represented using an array *data*. The least recently added item is at index f (front).

```
Input Parameters: None
Output Parameters: None
front() {
    return data[f]
}
```

Using a linked list to implement a stack

Algorithm 3.3.4 Initializing a Stack

This algorithm initializes a stack to empty. The stack is implemented as a linked list. The start of the linked list, referenced by t, is the top of the stack. An empty stack has `t = null`.

```
Input Parameters: None
Output Parameters: None
stack_init() {
    t = null
}
```

Algorithm 3.3.5 Testing for an Empty Stack

This algorithm returns true if the stack is empty or false if the stack is not empty. An empty stack has `t = null`.

```
Input Parameters: None
Output Parameters: None
empty() {
    return t == null
}
```

Algorithm 3.3.6 Adding an Element to a Stack

This algorithm adds the value *val* to a stack. The stack is implemented using a linked list. The start of the linked list, referenced by *t*, is the top of the stack.

```
Input Parameters: val
Output Parameters: None
push(val) {
    temp = new node
    temp.data = val
    temp.next = t
    t = temp
}
```

Algorithm 3.3.7 Removing an Element From a Stack

This algorithm removes the most recently added item from a stack. The stack is implemented using a linked list. The start of the linked list, referenced by t, is the top of the stack. The algorithm assumes that the stack is not empty.

```
Input Parameters: None
Output Parameters: None
pop() {
    t = t.next
}
```

Algorithm 3.3.8 Returning the Top Element in a Stack

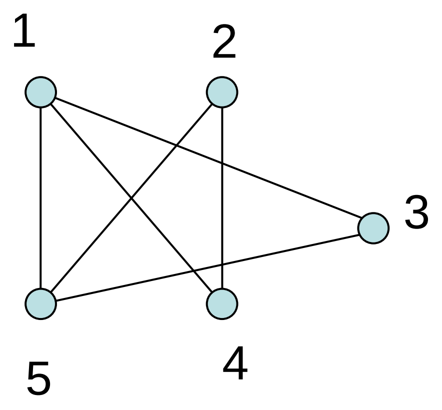
This algorithm returns, but does not remove, the most recently added item in a stack. The stack is implemented using a linked list. The start of the linked list, referenced by *t* , is the top of the stack. The algorithm assumes that the stack is not empty.

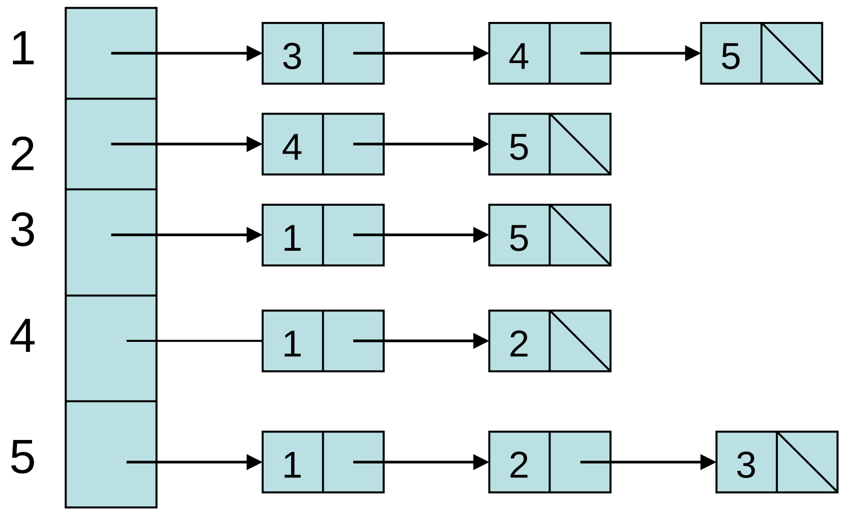
```
Input Parameters: None
Output Parameters: None
top() {
    return t.data
}
```

Adjacency Lists for Graphs

Linked lists can be used to represent graphs.

Example 4





Algorithm 3.3.10 Computing Vertex Degrees Using Adjacency Lists

This algorithm computes the degree of each vertex in a graph. The graph is represented using adjacency lists; $adj[i]$ is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex i. The *next* field in each node, except the last, references the next node in the list. The *next* field of the last node is null. The vertices are $1, 2, \ldots$.

Algorithm 3.3.10 Computing Vertex Degrees Using Adjacency Lists

```
Input Parameter: adj
Output Parameters: None
degrees1(adj) {
    for i = 1 to adj.last {
        count = 0
        ref = adj[i]
        while (ref != null) {
            count = count + 1
            ref = ref.next
        }
        println(“vertex ” + i + “ has degree ” + count)
    }
}
```

The Algorithm runs in time $\Theta(n+m)$, where n is the number of vertices and m number of edges in the graph.

Algorithm 3.3.11 Computing Vertex Degrees Using an Adjacency Matrix

This algorithm computes the degree of each vertex in a graph. The graph is represented using an adjacency matrix am , where $am[i][j]$ is 1 if there is an edge between i and j, and 0 if not. The vertices are $1, 2, \ldots$.

```
Input Parameter: am
Output Parameters: None
degrees2(am) {
    for i = 1 to am.last {
        count = 0
        for j = 1 to am.last
            if (am[i][j] == 1)
                count = count + 1
        println("vertex " + i + " has degree " + count)
    }
}
```

Algorithm 3.3.11 Computing Vertex Degrees Using an Adjacency Matrix

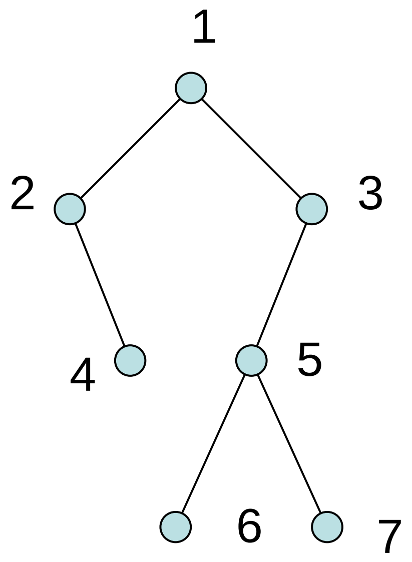
The Algorithm runs in time $\Theta(n^2)$, where n is the number of vertices in the graph.

Binary Trees

- A binary tree is a rooted tree where each node has either:
  - No children
  - One child
  - Two children.
- A child can be either a left child or a right child.
- To traverse a binary tree means to visit each node in the tree in some prescribed order.

Binary Trees

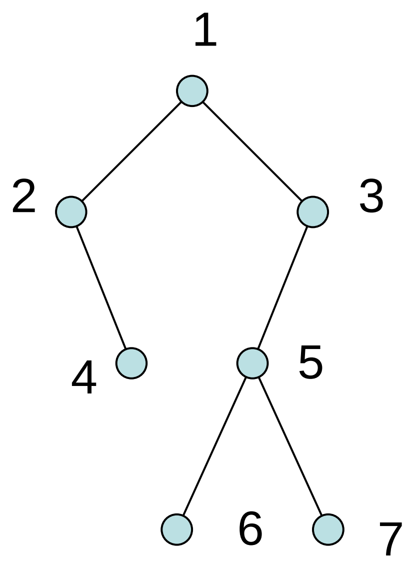
Example 5



Binary Trees

Preorder traversal of a binary rooted tree:

- If root is empty stop.
- Visit root
- Execute preorder on the binary tree rooted at the left child of the root
- Execute preorder on the binary tree rooted at the right child of the root



Algorithm 3.4.3 Preorder

This algorithm performs a preorder traversal of the binary tree with root *root*.

```
Input Parameter: root
Output Parameters: None
preorder(root)    {
    if (root != null) {
        // visit root
        preorder(root.left)
        preorder(root.right)
    }
}
```

Algorithm 3.4.5 Counting Nodes in a Binary Tree

This algorithm returns the number of nodes in the binary tree with root *root*.

```
Input Parameter: root
Output Parameters: None
count_nodes(root)
    if (root == null )
        return 0
    count = 1 // count root
    count = count + count_nodes(root.left)
        // add in nodes in left subtree
    count = count + count_nodes(root.right)
        // add in nodes in right subtree
    return count
}
```

Inorder traversal of a binary rooted tree:

- If root is empty stop.
- Execute inorder on the binary tree rooted at the left child of the root
- Visit root
- Execute inorder on the binary tree rooted at the right child of the root

Postorder traversal of a binary rooted tree:

- If root is empty stop.
- Execute postorder on the binary tree rooted at the left child of the root
- Execute postorder on the binary tree rooted at the right child of the root
- Visit root

Algorithm 3.4.9 Inorder

This algorithm performs an inorder traversal of the binary tree with root *root*.

```
Input Parameter: root
Output Parameters: None
inorder(root)    {
    if (root != null) {
        inorder(root.left)
        // visit root
        inorder(root.right)
    }
}
```

Algorithm 3.4.10 Postorder

This algorithm performs a postorder traversal of the binary tree with root *root*.

```
Input Parameter: root
Output Parameters: None
postorder(root)    {
    if (root != null) {
        postorder(root.left)
        postorder(root.right)
        // visit root
    }
}
```

Algorithm 3.4.14 Inserting Into a Binary Search Tree

This algorithm inserts the value *val* into a binary search tree with root *root*. If the tree is empty, *root* = null. The algorithm returns the root of the tree containing the added item. We assume that "new *node*" creates a new node with data field *data* and reference fields *left* and *right*.

```
Input Parameter: root,val
Output Parameters: None
BSTinsert(root,val) {
    // set up node to be added to tree
    temp = new node
    temp.data = val
    temp.left = temp.right = null
    if (root == null) // special case: empty tree
        return temp
    BSTinsert_recurs(root,temp)
    return root
}
...
```

```
...
BSTinsert_recurs(root,temp) {
    if (temp.data  $\leq$  root.data)

        if (root.left == null )
            root.left ==  temp
        else
            BSTinsert_recurs(root.left,temp)
    else
        if (root.right == null)
            root.right == temp
        else
            BSTinsert_recurs(root.right,temp)
}
```

Algorithm 3.4.15 Deleting from a Binary Search Tree, Special Case

This algorithm deletes the node referenced by *ref* from a binary search tree with root *root*. The node referenced by *ref* has zero children or exactly one child. We assume that each node N in the tree has a parent field, *N.parent*. If N is the root, *N.parent* is null. The algorithm returns the root of the tree that results from deleting the node.

```
Input Parameter: root,ref
Output Parameters: None
BSTreplace(root,ref) {
    // set child to ref's child, or null, if no child
    if (ref.left == null)
        child = ref.right
    else
        child = ref.left
    if (ref == root) {
        if (child != null)
            child.parent = null
        return child
    }
    if (ref.parent.left == ref) // is ref left child?
        ref.parent.left = child
    else
        ref.parent.right = child
    if (child != null)
        child.parent = ref.parent
    return root
}
```

Algorithm 3.4.16 Deleting from a Binary Search Tree

This algorithm deletes the node referenced by *ref* from a binary search tree with root *root*. We assume that each node *N* in the tree has a parent field, *N.parent*. If *N* is the root, *N.parent* is null. The algorithm returns the root of the tree that results from deleting the node.

```
Input Parameter: root,val
Output Parameters: None
BSTdelete(root,ref) {
    // if zero or one children, use Algorithm 3.4.15
    if (ref.left == null || ref.right == null)
        return BSTreplace(root,ref)
    // find node succ containing a minimum data item in ref’s
    succ = ref.right                              // right subtree
    while (succ.left != null)
        succ = succ.left
    // “move” succ to ref, thus deleting ref
    ref.data = succ.data
    // delete succ
    return BSTreplace(root,succ)
}
```

Priority Queues

A priority queue is an abstract data type that allows:

- insertion of items with specified priorities, and
- deletion of an item with the highest priority

Example 6

Suppose we insert items with priorities 20, 389, 12 and 7 into initially empty priority queue. After deleting an item, the priority queue contains 20, 12, 7 and after inserting 67 it contains 20, 12, $7, 67$.

Implementing a Priority Queue Using an Array

Insertion (at the end of the array)	$\Theta(1)$
Deletion	
– Finding an item with largest priority	$\Theta(n)$
– Shifting all elements to the right one cell to the left	$\mathrm{O(n)}$ ($\Theta(n)$?)
– TOTAL	$\Theta(n)$

- For n insertions and deletions we have $\mathrm{O}(n^2)$ (is it also $\Theta(n^2)$?)

Implementing a Priority Queue Using an Array

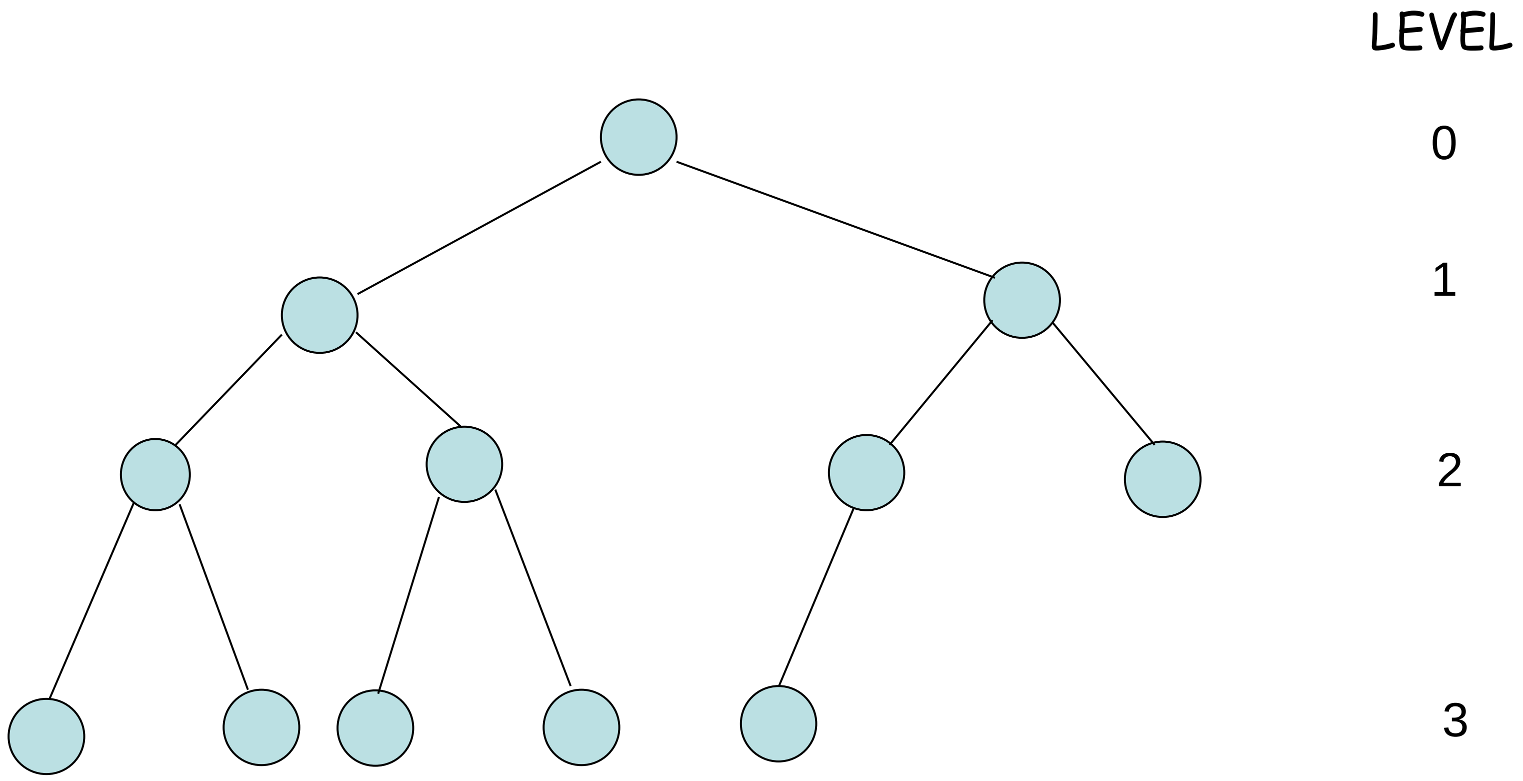
What happens if we maintain a sorted array?

Insertion	
– Locating the position	$\Theta(\lg n)$
– Shifting all elements to the right one cell to the right	$\Theta(n)$
Deletion	$\Theta(1)$
– TOTAL	$\Theta(n)$

For n insertions and deletions, we have $\Theta(n^2)$.

Heaps

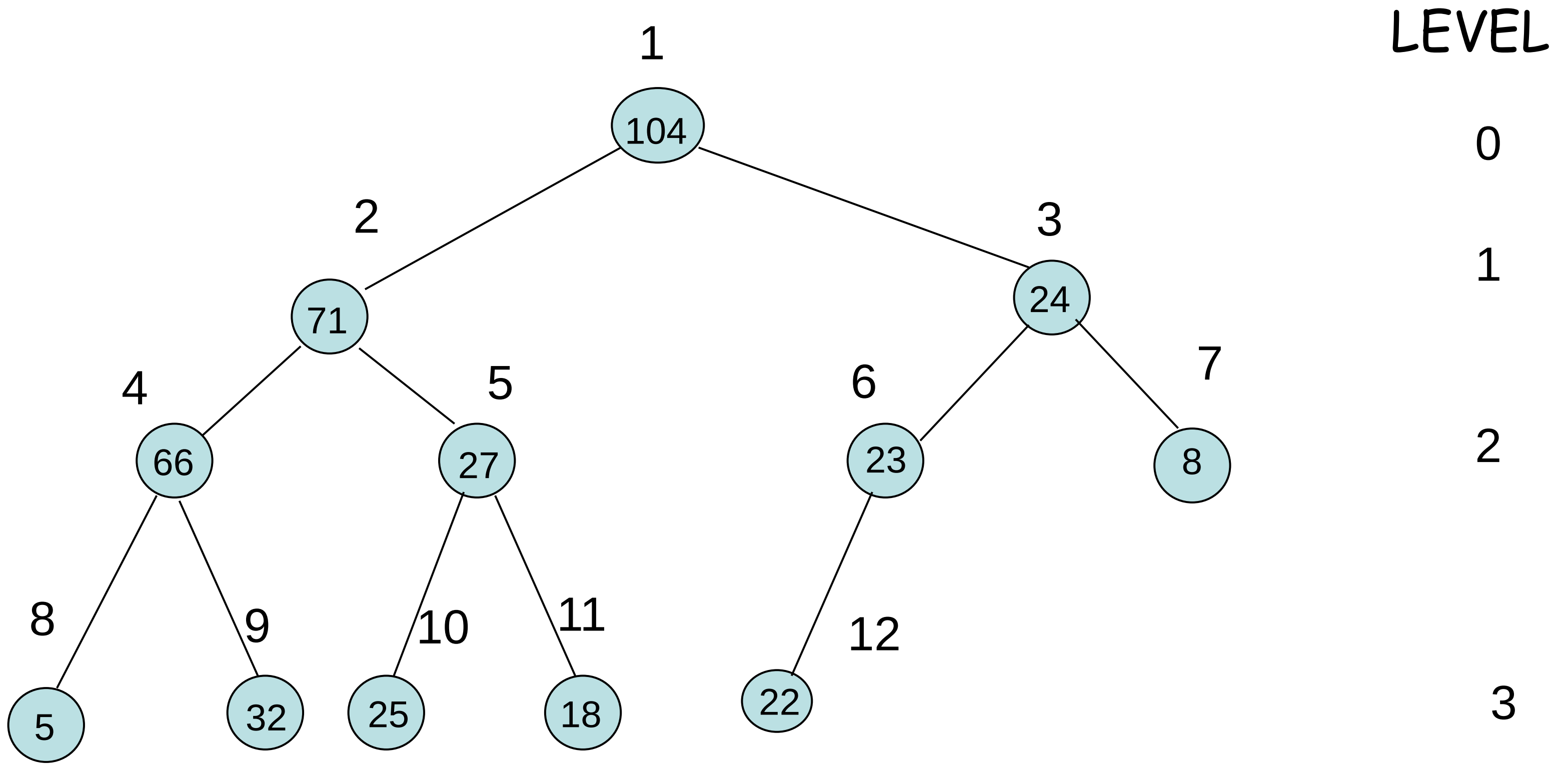
A heap is a binary tree where all levels except possibly the last level are completely filled in; the nodes in the last level are at the left.



Binary Maxheaps

In a binary maxheap a value of each node is greater or equal to the values of its children.

Example 7



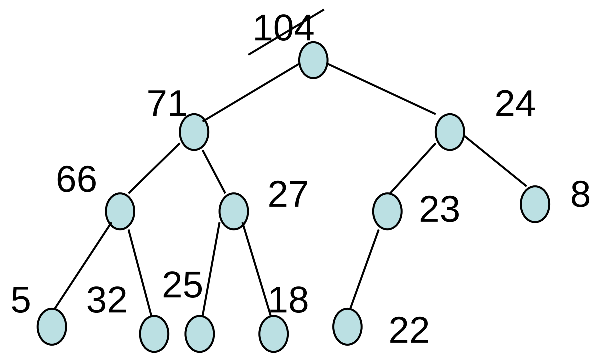
104	71	24	66	27	23	8	5	32	25	18	22
1	2	3	4	5	6	7	8	9	10	11	12

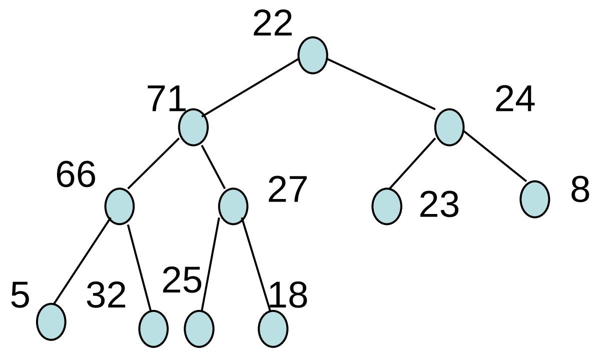
Algorithm 3.5.6 Largest

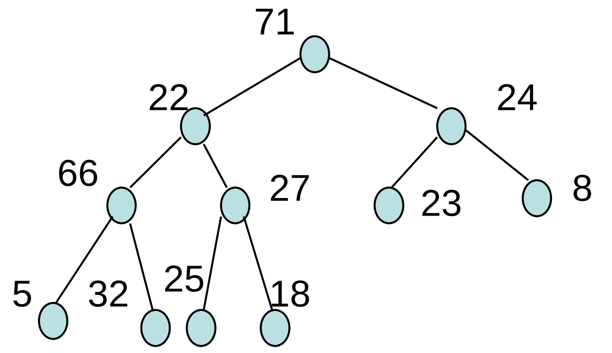
This algorithm returns the largest value in a heap. The array v represents the heap.

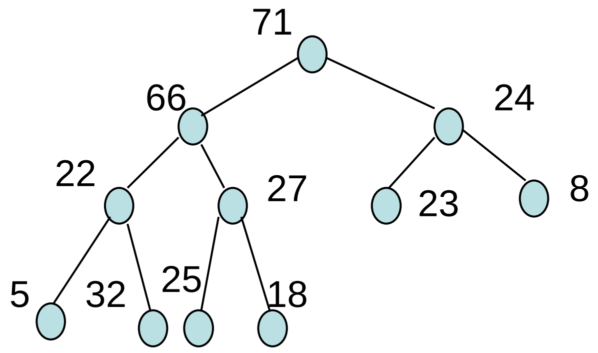
```
Input Parameter: v
Output Parameters: None
heap_largest(v) {
    return v[1]
}
```

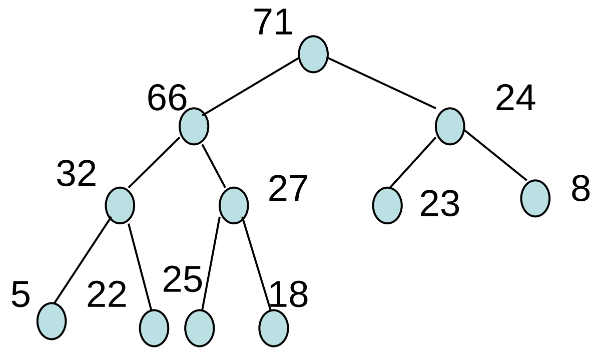
Deleting from a Heap











Algorithm 3.5.7 Siftdown

The array v represents a heap structure indexed from 1 to n.

The left and right subtrees of node i are heaps.

After *siftdown*(v, i, n) is called, the subtree rooted at i is a heap.

```
Input Parameter: v,i,n
Output Parameters: v
siftdown(v,i,n) {
    temp = v[i]
    // 2 * i ≤ n tests for a left child
    while (2 * i ≤ n) {
        child = 2 * i
        // if there is a right child and it is
        // bigger than the left child, move child
        if (child < n && v[child + 1] > v[child])
            child = child + 1
        // move child up?
        if (v[child] > temp)
            v[i] = v[child]
        else
            break // exit while loop
        i = child
    }
    // insert original v[i] in correct spot
    v[i] = temp
}
```

Algorithm 3.5.9 Delete

This algorithm deletes the root (the item with largest value) from a heap containing n elements. The array v represents the heap.

```
Input Parameters: v,n
Output Parameters: v,n
heap_delete(v,n) {
    v[1] = v[n]
    n = n - 1
    siftdown(v,1,n)
}
```

Complexity: $\Theta(\lg n)$

Algorithm 3.5.10 Insert

This algorithm inserts the value *val* into a heap containing n elements. The array v represents the heap.

```
Input Parameters: val,v,n
Output Parameters: v,n
heap_insert(val,v,n) {
    i = n = n + 1
    // i is the child and i/2 is the parent.
    // If i > 1, i is not the root.
    while (i > 1 && val > v[i/2]) {
        v[i] = v[i/2]
        i = i/2
    }
    v[i] = val
}
```

Complexity?

Algorithm 3.5.12 Heapify

This algorithm rearranges the data in the array v, indexed from 1 to n, so that it represents a heap.

```
Input Parameters: $v,n$
Output Parameters: $v$
heapify($v,n$) {
    // $n/2$ is the index of the parent of the last node
    for $i$ = $n/2$ downto 1
        siftdown($v,i,n$)
}
```

Complexity?

Algorithm 3.5.12 Heapify

Complexity? $\Theta(n)$

Proof:

- The siftdown is applied to all the nodes except the leaves.
- For a node at the level $h-1$ the siftdown takes at most 1 step.
- In general, for a node at level i, the siftdown takes at most $h-i$ steps.
- There are
    - 2^i internal nodes on level i, $i < h-1$
    - at most 2^{h-1} internal nodes on level $h-1$
- Therefore, the worst-case time

$$T(n) \leq 1 \times 2^{h-1} + 2 \times 2^{h-2} + 3 \times 2^{h-3} + \ \ldots + h \times 2^{h-h}$$

$$T(n) \leq \sum_{i=1}^{h} i \times 2^{h-i} = 2^h \sum_{i=1}^{h} \frac{i}{2^i} < 2^h \times 2 = 2^{\lfloor \lg n \rfloor} \times 2 \leq 2^{\lg n} \times 2 = 2n$$

Therefore, $T(n) = \mathrm{O}(n)$.

Algorithm 3.5.12 Heapify

We need to show that

$$\sum_{i=1}^{h} \frac{i}{2^i} < 2$$

					total
$\frac{1}{2}$	$\frac{1}{2^2}$	$\frac{1}{2^3}$	$\cdots$	$\frac{1}{2^h}$	< 1
	$\frac{1}{2^2}$	$\frac{1}{2^3}$	$\cdots$	$\frac{1}{2^h}$	$< \frac{1}{2}$
		$\frac{1}{2^3}$	$\cdots$	$\frac{1}{2^h}$	$< \frac{1}{2^2}$
			$\cdots$	$\cdots$	$\cdots$
				$\frac{1}{2^h}$	$< \frac{1}{2^{h-1}}$
TOTAL					< 2

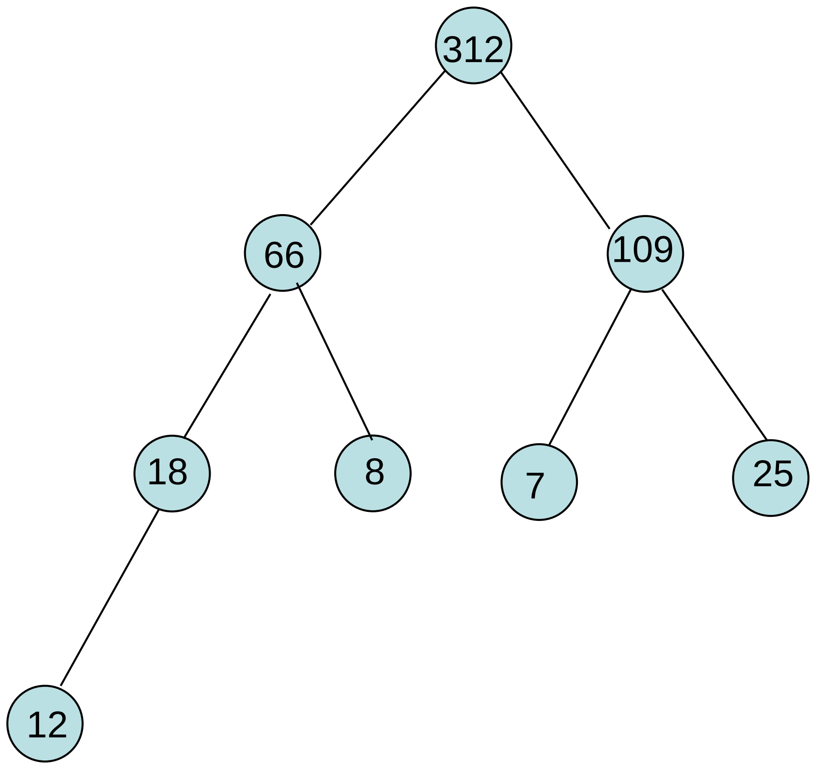
Algorithm 3.5.12 Heapify

On the other hand, there are $\frac{n}{2}$ internal nodes (non-leaves) in the heap, therefore the complexity is $\Omega(n)$; thus, we have $\Theta(n)$.

Note that this proves not only that the worst-case complexity is $\Theta(n)$, but also the best and average case complexity.

Indirect Heaps

Example 8



key

66	12	312	25	8	109	7	18
1	2	3	4	5	6	7	8

into

2	8	1	7	5	3	6	4

outof

3	1	6	8	5	7	4	2

Algorithm 3.5.15 Increase

This algorithm increases a value in an indirect heap and then restores the heap. The input is an index i into the *key* array, which specifies the value to be increased, and the replacement value *newval*.

```
Input Parameters: $i$,$newval$
Output Parameters: None
$increase$($i$,$newval$) {
    $key$[$i$] = $newval$
    // $p$ is the parent index in the heap structure
    // $c$ is the child index in the heap structure
    $c$ = $into$[$i$]
    $p$ = $c$/2
    while ($p \geq 1$) {
        if ($key$[$outof$[$p$]] $\geq$ $newval$)
            break // exit while loop
        // move value at $p$ down to $c$
        $outof$[$c$] = $outof$[$p$]
        $into$[$outof$[$c$]] = $c$
        // move $p$ and $c$ up
        $c$ = $p$
        $p$ = $c$/2
    }
    // “put” $newval$ in heap structure at index $c$
    $outof$[$c$] = $i$
    $into$[$i$] = $c$
}
```

Algorithm 3.5.16 Heapsort

This algorithm sorts the array $v[1], \ldots, v[n]$ in nondecreasing order. It uses the *siftdown* and *heapify* algorithms (Algorithms 3.5.7 and 3.5.12).

```
Input Parameters: v,n
Output Parameter: v
heapsort(v,n)  {
    // make v into a heap
    heapify(v,n)
    for i = n downto 2 {
        // v[1] is the largest among v[1], ... , v[i].
        // Put it in the correct cell.
        swap(v[1],v[i])
        // Heap is now at indexes 1 through i - 1.
        // Restore heap.
        siftdown(v,1,i - 1 )
    }
}
```

Disjoint sets

We consider *nonempty pairwise disjoint sets*, where each set has an element marked as a representative element of the set.

Sets X and Y are *disjoint* if

$$X \cap Y = \emptyset$$

Example 9

The sets $\{\underline{\mathbf{1}}\}\{2, \underline{\mathbf{4}}, 3\}\{5, \underline{\mathbf{6}}, 7\}$ are nonempty pairwise disjoint sets. The mark element in each set is represented in bold and underlined.

Disjoint sets

The operations:

- `makeset(i)` - constructs the set $\{i\}$
- `findset(i)` - returns the marked member of the set to which i belongs
- `union(i,j)` - replaces the set containing i and the set containing j with their union

Example 10

makeset (1)

makeset (2)

makeset (3)

makeset (4)

makeset (5)

makeset (6)

makeset (7)

We have

$$\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}$$

union (2,4)

union (2,3)

union (5,6)

union (6,7)

We have

$$\{\underline{\mathbf{1}}\}\{2,\underline{\mathbf{4}},3\}\{5,\underline{\mathbf{6}},7\}$$

findset(2) returns 4.

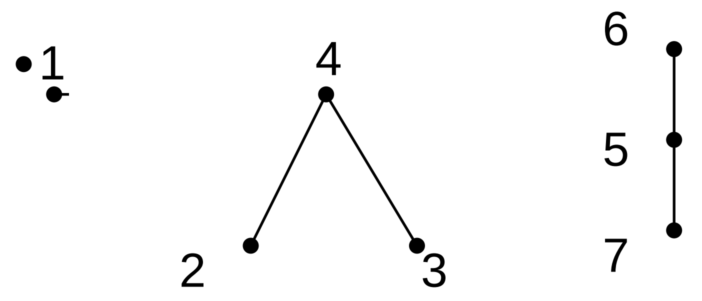
Disjoint sets

We represent the disjoint-set abstract data type as an arbitrary tree with the marked element as a root.

Note that such a tree is not necessarily binary.

<u>Example 11</u>

Sets $\{\underline{\mathbf{1}}\}\{2,\underline{\mathbf{4}},3\}\{5,\underline{\mathbf{6}},7\}$ may be represented as follows (note that there is more than one way to represent these sets):



Algorithm 3.6.4 Makeset, Version 1

This algorithm represents the set $\{i\}$ as a one-node tree.

```
Input Parameter: i
Output Parameters: None
makeset1(i) {
   parent[i] = i
}
```

Algorithm 3.6.5 Findset, Version 1

This algorithm returns the root of the tree to which *i* belongs.

```
Input Parameter: i
Output Parameters: None
findset1(i) {
    while (i != parent[i])
        i = parent[i]
    return i
}
```

Algorithm 3.6.6 Mergetrees, Version 1

This algorithm receives as input the roots of two distinct trees and combines them by making one root a child of the other root.

```
Input Parameters: i,j
Output Parameters: None
mergetrees1(i,j) {
    parent[i] = j
}
```

Algorithm 3.6.8 Union, Version 1

This algorithm receives as input two arbitrary values i and j and constructs the tree that represents the union of the sets to which i and j belong. The algorithm assumes that i and j belong to different sets.

```
Input Parameters: i,j
Output Parameters: None
union1(i,j) {
    mergetrees1(findset1(i), findset1(j))
}
```

Q: What is a height of a tree constructed by previous algorithms?

A: At most $n-1$, where n is the number of elements in the set.

It is more desirable for a tree representing a disjoint set to have a smaller height as then algorithms findset and union would be more efficient.

The following algorithms guarantee that the height of a tree with n nodes is at most $\lfloor \lg n \rfloor$.

Algorithm 3.6.9 Makeset, Version 2

This algorithm represents the set $\{i\}$ as a one-node tree and initializes its height to 0.

```
Input Parameter: i
Output Parameters: None
makeset2(i) {
    parent[i] = i
    height[i] = 0
}
```

Algorithm 3.6.10 Findset, Version 2

This algorithm returns the root of the tree to which *i* belongs.

```
Input Parameter: $i$
Output Parameters: None
$findset2$($i$) {
    while ($i$ != $parent$[$i$])
        $i$ = $parent$[$i$]
    return $i$
}
```

Algorithm 3.6.11 Mergetrees, Version 2

This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller height a child of the other root. If the trees have the same height, we arbitrarily make the root of the first tree a child of the other root.

```
Input Parameters: i,j
Output Parameters: None
mergetrees2(i,j) {
    if (height[i] < height[j])
        parent[i] = j
    else if (height[i] > height[j])
        parent[j] = i
    else {
        parent[i] = j
        height[j] = height[j] + 1
    }
}
```

Algorithm 3.6.12 Union, Version 2

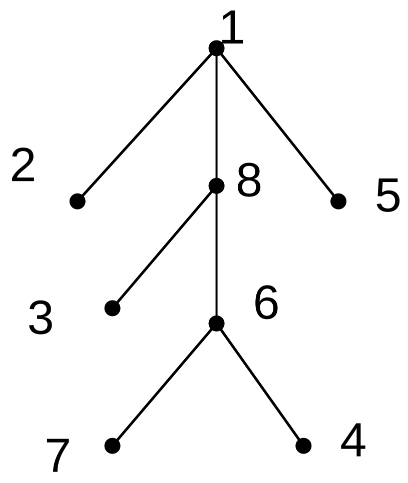
This algorithm receives as input two arbitrary values i and j and constructs the tree that represents the union of the sets to which i and j belong. The algorithm assumes that i and j belong to different sets.

```
Input Parameters: i,j
Output Parameters: None
union2(i,j) {
    mergetrees2(findset2(i), findset2(j))
}
```

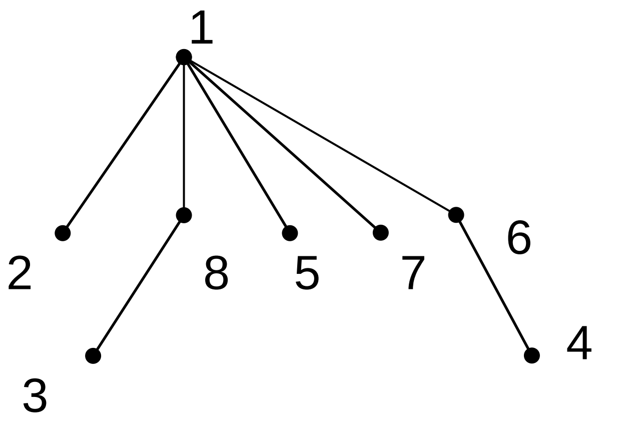
Can we do better than this?

We can apply the so-called path compression in the *findset* algorithm. When executing *findset(i)* , we make every node on the path from *i* to the root a child of the root (except, of course, the root itself).

Example 12



Original tree



The tree after *findset(7)* is called

The following algorithms incorporate path compression. Note that the algorithms use “rank” instead of “height”. As the path compression can potentially decrease the height if the tree, this parameter does not necessarily denote the height of the tree. It rather denotes an upper bound on its height, which we call “rank”.

Algorithm 3.6.16 Makeset, Version 3

This algorithm represents the set $\{i\}$ as a one-node tree and initializes its rank to 0.

```
Input Parameter: i
Output Parameters: None
makeset3(i) {
    parent[i] = i
    rank[i] = 0
}
```

Algorithm 3.6.17 Findset, Version 3

This algorithm returns the root of the tree to which i belongs and makes every node on the path from i to the root, except the root itself, a child of the root.

```
Input Parameter: i
Output Parameters: None
findset3(i) {
    root = i
    while (root != parent[root])
        root = parent[root]
    j = parent[i]
    while (j != root) {
        parent[i] = root
        i = j
        j = parent[i]
    }
    return root
}
```

Algorithm 3.6.18 Mergetrees, Version 3

This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller rank a child of the other root. If the trees have the same rank, we arbitrarily make the root of the first tree a child of the other root.

```
Input Parameters: i,j
Output Parameters: None
mergetrees3(i,j) {
    if (rank[i] < rank[j])
        parent[i] = j
    else if (rank[i] > rank[j])
        parent[j] = i
    else {
        parent[i] = j
        rank[j] = rank[j] + 1
    }
}
```

Algorithm 3.6.19 Union, Version 3

This algorithm receives as input two arbitrary values *i* and *j* and constructs the tree that represents the union of the sets to which *i* and *j* belong. The algorithm assumes that *i* and *j* belong to different sets.

```
Input Parameters: i,j
Output Parameters: None
union3(i,j) {
    mergetrees3(findset3(i), findset3(j))
}
```