CMP_SC 3050: Elementary Data structures

Rohit Chadha

September 3, 2014

Data structures

- A data structure is a structure to organize data
  - By data, we mean input, output and intermediate data used to compute the output
- Efficiency of algorithms depends on choosing the right data structure for the computational problem the algorithm is solving
- Some common data structures
  - Arrays
  - Linked lists
  - Records
  - Stacks
  - Queues
  - Binary trees
  - Heaps
  - And many more ...

Stack

A stack is a sequence of elements which supports the following operations:

PUSH: Inserts an element to the front of the sequence

POP: Delete an element from the front of the sequence

EMPTY: Checks if the sequence is empty

Elements are extracted in last-in-first-out (LIFO) order, i.e., elements are picked in the reverse order of when they were inserted.

Implementing Stacks

- A stack of size n can be implemented as an array $S[1\,..\,n]$
- A stack S has an attribute $S.top$ that indexes the most recently inserted element
- $S.top = 0$ iff the stack is empty

Example: Stack S with 4 elements $9, 2, 6, 15$

...	
	7
	6
	5
9	$\leftarrow S.top = 4$
2	3
6	2
15	1

Stack operations

STACK-EMPTY(S)

if $S.top == 0$
    return TRUE
else return FALSE

Stack operations

STACK-EMPTY(S)

if $S.top == 0$
    return TRUE
else return FALSE

PUSH(S, x)

if $S.top == n$
    error "overflow"
else $S.top = S.top + 1$
    $S[S.top] = x$

Stack operations

STACK-EMPTY(S)

if $S.top == 0$

  return TRUE

else return FALSE

PUSH(S, x)

if $S.top == n$

  error "overflow"

else $S.top = S.top + 1$

  $S[S.top] = x$

POP(S)

if $S.top == 0$

  error "underflow"

else $S.top = S.top - 1$

  return $S[S.top + 1]$

Stack operations

STACK-EMPTY(S)
  if $S.top == 0$
    return TRUE
  else return FALSE

POP(S)
  if $S.top == 0$
    error "underflow"
  else $S.top = S.top - 1$
    return $S[S.top + 1]$

PUSH(S, x)
  if $S.top == n$
    error "overflow"
  else $S.top = S.top + 1$
    $S[S.top] = x$

- All the stack operations are $O(1)$ with this implementation (i.e., constant time)

Stack operations

STACK-EMPTY(S)

if $S.top == 0$

  return TRUE

else return FALSE

POP(S)

if $S.top == 0$

  error "underflow"

else $S.top = S.top - 1$

  return $S[S.top + 1]$

PUSH(S, x)

if $S.top == n$

  error "overflow"

else $S.top = S.top + 1$

  $S[S.top] = x$

- All the stack operations are $O(1)$ with this implementation (i.e., constant time)
- Never forget the corner cases!

Example

Stack S with 4 elements $9, 2, 6, 15$

...	
	7
	6
	5
9	$\leftarrow S.top = 4$
2	3
6	2
15	1

$\text{PUSH}(S, 17)$ followed by $\text{PUSH}(S, 3)$

Example

Stack S with 4 elements $9, 2, 6, 15$

$\dots$	
	7
	6
	5
9	$\leftarrow S.top = 4$
2	3
6	2
15	1

$\textsc{Push}(S, 17)$ followed by $\textsc{Push}(S, 3)$

$\dots$	
	7
3	$\leftarrow S.top = 6$
17	5
9	4
2	3
6	2
15	1

Example continued

Stack S with elements $3, 17, 9, 2, 6, 15$

...	
	7
3	$\leftarrow S.top = 6$
17	5
9	4
2	3
6	2
15	1

Popping results in

Example continued

Stack S with elements $3, 17, 9, 2, 6, 15$

...	
	7
3	$\leftarrow S.top = 6$
17	5
9	4
2	3
6	2
15	1

Popping results in

...	
	7
3	6
17	$\leftarrow S.top = 5$
9	4
2	3
6	2
15	1

Example continued

Stack S with elements $3, 17, 9, 2, 6, 15$

...	
	7
3	$\leftarrow S.top = 6$
17	5
9	4
2	3
6	2
15	1

Popping results in

...	
	7
3	6
17	$\leftarrow S.top = 5$
9	4
2	3
6	2
15	1

Be careful, $S[6]$ has some value which is now meaningless!

Queue

A queue is a sequence of elements which supports the following operations:

- **ENQUEUE**: Inserts an element to the back of the sequence
- **DEQUEUE**: Delete an element from the front of the sequence
- **EMPTY**: Checks if the sequence is empty

Elements are extracted in First-in-first-out (FIFO) order, i.e., elements are picked in the same order of when they were inserted.

Queue

A queue is a sequence of elements which supports the following operations:

ENQUEUE:	Inserts an element to the back of the sequence
DEQUEUE:	Delete an element from the front of the sequence
EMPTY:	Checks if the sequence is empty

Elements are extracted in First-in-first-out (FIFO) order, i.e., elements are picked in the same order of when they were inserted.

- Can be implemented with arrays with all operations $O(1)$

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050
  - Each element of a linked list L has at least two attributes
    1. *key* which contains data such as integers
    2. *next* pointer which points to the next element in the list

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050
  - Each element of a linked list L has at least two attributes
    1. *key* which contains data such as integers
    2. *next* pointer which points to the next element in the list
  - There might be some other attributes also

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050
  - Each element of a linked list L has at least two attributes
    1. *key* which contains data such as integers
    2. *next* pointer which points to the next element in the list
  - There might be some other attributes also
  - For a list L, the attribute $L.head$ points to the first element of the list

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050
  - Each element of a linked list L has at least two attributes
    1. *key* which contains data such as integers
    2. *next* pointer which points to the next element in the list
  - There might be some other attributes also
  - For a list L, the attribute $L.head$ points to the first element of the list
  - L is empty iff $L.head == \textsc{Nil}$

Linked Lists

- A linked list is a data structure in which objects are arranged in a linear order
- Unlike the array, the linear order is determined by a pointer in each object
- We have seen singly linked lists in CMP_SC 2050
  - Each element of a linked list L has at least two attributes
    1. *key* which contains data such as integers
    2. *next* pointer which points to the next element in the list
  - There might be some other attributes also
  - For a list L, the attribute $L.head$ points to the first element of the list
  - L is empty iff $L.head == \textsc{Nil}$
- Stacks and queues can be implemented as singly linked lists

Doubly-linked lists

Doubly-linked lists

- Each element of a linked list L has at least three attributes
  1. *key* which contains data such as integers
  2. *next* pointer which points to the next element in the list
  3. *prev* pointer which points to the previous element in the list

Doubly-linked lists

- Each element of a linked list L has at least three attributes
  1. *key* which contains data such as integers
  2. *next* pointer which points to the next element in the list
  3. *prev* pointer which points to the previous element in the list

Some standard operations on lists

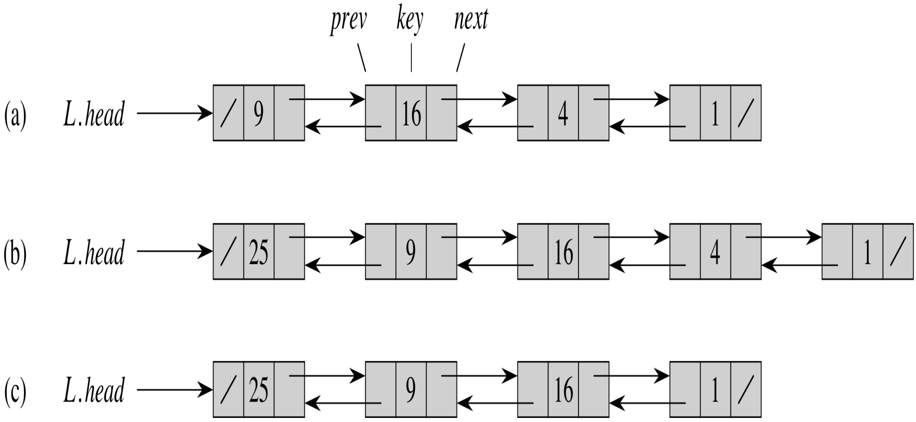
LIST-SEARCH: Searches for an element in the list

LIST-INSERT: Inserts an element at the beginning of the list

LIST-DELETE: Deletes a specified element from the list

LIST-EMPTY: Checks if the list is empty

Example



(a) A doubly linked list L

(b) The result of inserting 25 to the list in (a)

(c) Result of deleting 4 from the list in (b)

Algorithm for searching in a doubly-linked list

$\textsc{List-Search}(S, k)$

$x = L.head$
while $x \neq \textsc{Nil}$ and $x.key \neq k$
  $x = L.next$
return x

- Runs in $\Theta(n)$ time

Algorithm for searching in a doubly-linked list

```
List-Search(S, k)
    x = L.head
    while x ≠ Nil and x.key ≠ k
        x = L.next
    return x
```

- Runs in $\Theta(n)$ time
- Algorithms for inserting and deleting from the list?

Binary Trees

Binary Trees

A binary tree T is a data structure defined on a finite collection of nodes such that

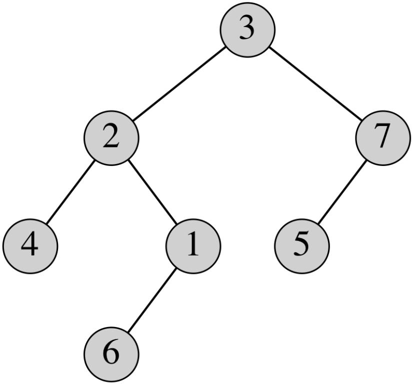
- Either the collection is empty (also called the NIL) tree
- or the nodes can be divided into three disjoint sets
  - A root node
  - A binary tree called its left subtree
  - A binary tree called its right subtree

Binary Trees

A binary tree T is a data structure defined on a finite collection of nodes such that

- Either the collection is empty (also called the NIL) tree
- or the nodes can be divided into three disjoint sets
  - A root node
  - A binary tree called its left subtree
  - A binary tree called its right subtree

Example



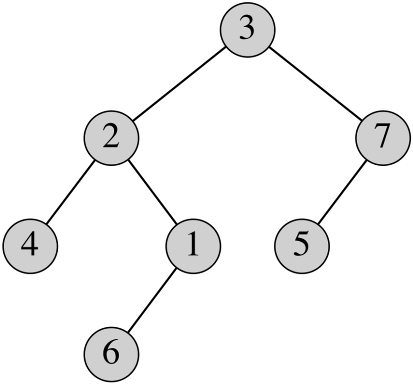
Binary Trees

A binary tree T is a data structure defined on a finite collection of nodes such that

- Either the collection is empty (also called the NIL) tree
- or the nodes can be divided into three disjoint sets
  - A root node
  - A binary tree called its left subtree
  - A binary tree called its right subtree

This is a recursive definition!

Example



Binary Trees continued

Left child: Left child of a node is the root of the left subtree

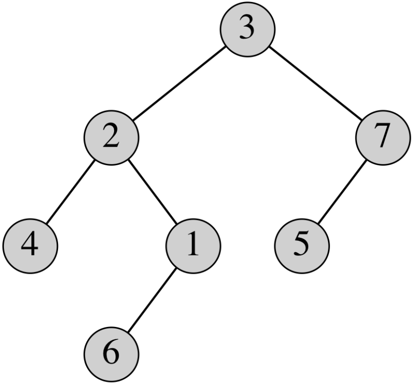
Right child: Right child of a node is the root of the right subtree

Parent: A node n_1 is a parent of n_2 if n_2 is a child of n_1

Degree: Degree of a node is the number of its children
Can be 0, 1 or 2

Leaf node: A node n is a leaf node if its degree is 0

Internal node: A node n is a leaf node if its degree is > 0

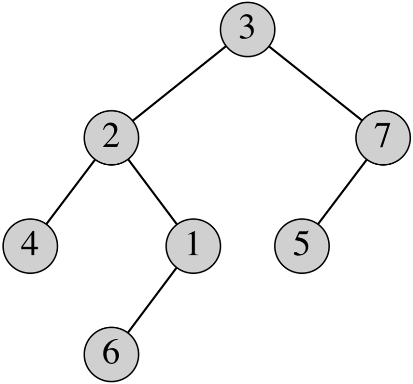


Binary Trees continued

Depth: Depth of a node is its distance from the root

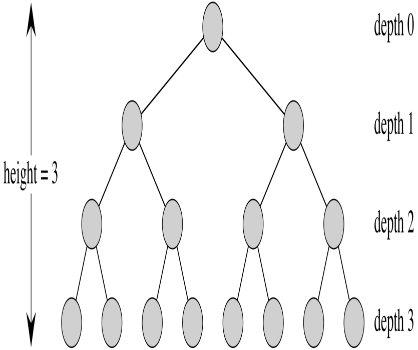
Height: Height of a node is the maximum distance from the node to a leaf in its subtree

Height of the tree: is the height of the root



Complete Binary Trees

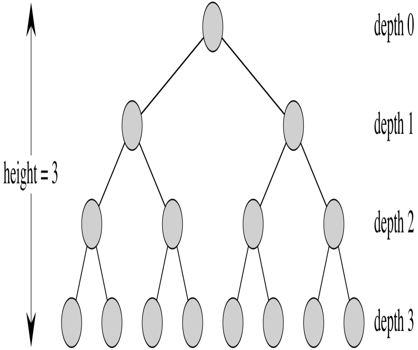
- A complete binary tree is a tree in which all leaves have the same depth and all internal nodes have degree 2



Complete Binary Trees

- A complete binary tree is a tree in which all leaves have the same depth and all internal nodes have degree 2
- A complete tree of height h has
  - 1 node at depth 0
  - 2 nodes at depth 1
  - 4 nodes at depth 2
  - ...
  - 2^h nodes at depth h

  Thus a complete tree of height h has 2^h leaf nodes

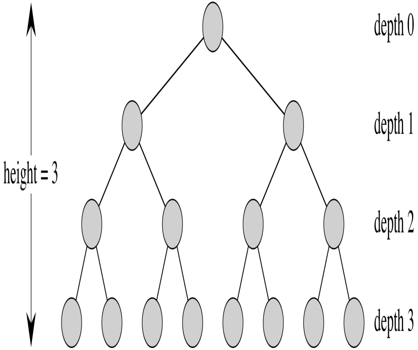


Complete Binary Trees

- A complete binary tree is a tree in which all leaves have the same depth and all internal nodes have degree 2
- A complete tree of height h has
  - 1 node at depth 0
  - 2 nodes at depth 1
  - 4 nodes at depth 2
  - ...
  - 2^h nodes at depth h

  Thus a complete tree of height h has 2^h leaf nodes
- The total number of nodes of a complete tree of height h is

$$1 + 2 + 2^2 \cdots + 2^h = 2^{h+1} - 1$$

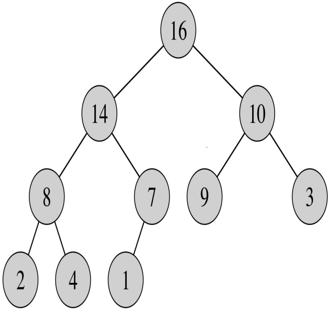


Heaps

A heap is a nearly complete binary tree:

- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

Example



Heaps

A heap is a nearly complete binary tree:

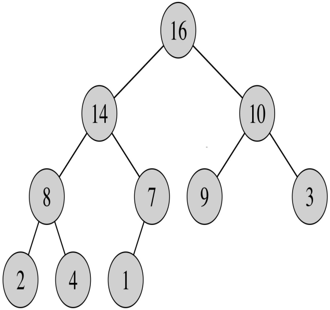
- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

Example



Heaps

A heap is a nearly complete binary tree:

- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

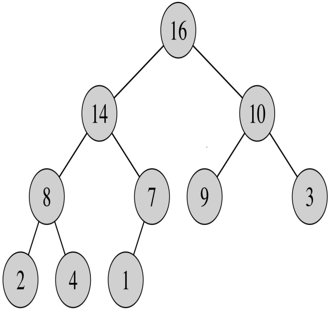
In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

2. Min-heap property: The value stored at every node must be less than the value stored in its children

Example



Heaps

A heap is a nearly complete binary tree:

- The tree must be completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

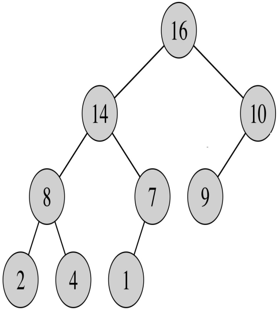
In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

2. Min-heap property: The value stored at every node must be less than the value stored in its children

Not a heap:



Heaps

A heap is a nearly complete binary tree:

- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

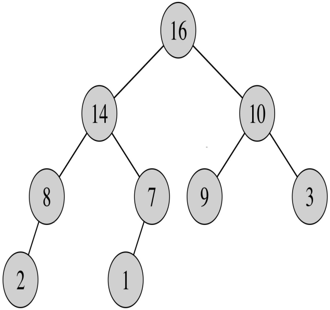
In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

2. Min-heap property: The value stored at every node must be less than the value stored in its children

Not a heap:



Heaps

A heap is a nearly complete binary tree:

- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

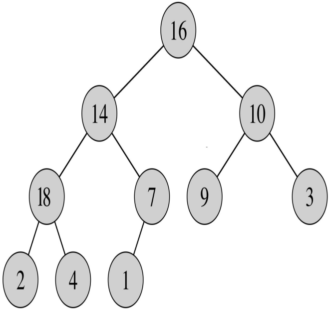
In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

2. Min-heap property: The value stored at every node must be less than the value stored in its children

Not a heap:



A question about heaps

What is the height of a heap with n nodes?

$$\Theta(log n)$$

A question about heaps

What is the height of a heap with n nodes?

$$\Theta(log n)$$

Please read Sections 10.1, 10.2 and Appendix B.5 from the book.
We have started Chapter 6.