Introduction to Computer Science
Lecture 7: Data Abstractions

Department of Electrical Engineering
National Taiwan University

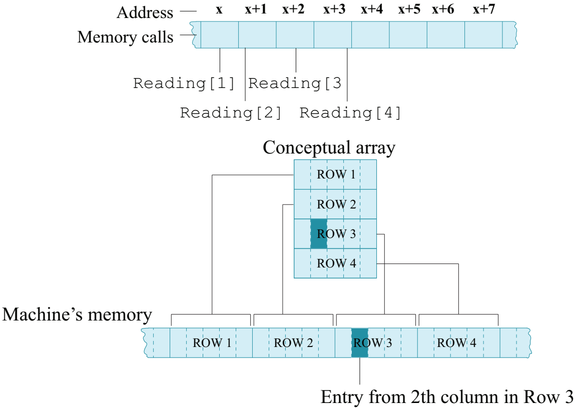
Data Structure Fundamentals

- Arrays
  - Homogeneous
  - Heterogeneous
- Lists
  - Storage
    - Contiguous lists (arrays)
    - Linked lists
  - Operations
    - Stacks: FILO
    - Queues: FIFO
- Trees
  - Binary search tree
- Binary heaps

Data Structure Concepts

- Abstraction vs. real data
- Dynamic vs. static structures
- Pointers: locating data
    - Program counter → instruction pointer
- Data structure implementation

Homogeneous Arrays



Address polynomial: $x + c \cdot (i - 1) + (j - 1)$

High-dimensional arrays: array of (array of . . .) arrays.

Array Addresses

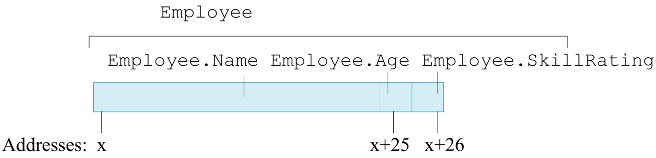
$x = 1000$	$x[0] = 1000$	$x[0][0]$ 1000	$x[0][1]$ 1004	$x[0][2]$ 1008	$x[0][3]$ 1012
	$x[1] = 1016$	$x[1][0]$ 1016	$x[1][1]$ 1020	$x[1][2]$ **1024**	$x[1][3]$ 1028
	$x[2] = 1032$	$x[2][0]$ 1032	$x[2][1]$ 1036	$x[2][2]$ 1040	$x[2][3]$ 1044

int x[3][4]

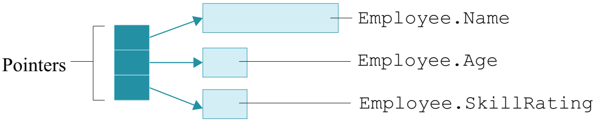
$$x[1][2] \rightarrow 1000 + 1*4*4 + 2*4 \rightarrow 1016\ (x[1]) + 8 \rightarrow 1024$$

sizeof(int): 4

Heterogeneous Arrays



a. Array stored in a contiguous block



b. Array components stored in separate locations

Using pointers to locate separate data

Template Functions & Classes

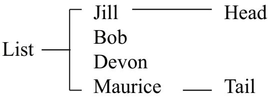
```
int Add(const int a, const int b) {
    return a+b;
}
```

```
template <class T>
T Add(const T& a, const T& b) {
    return (a+b);
}
```

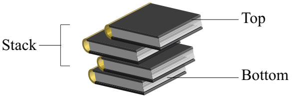
```
Complex<double> var;
```

```
template <class T>
class Complex {
public:
    Complex (const T&, const T&);
private:
    T re;
    T im;
};
```

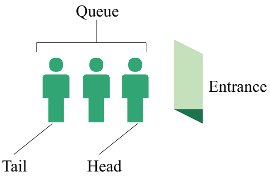
Lists, Stacks, and Queues



a. A list of names



b. A stack of books



c. A queue of people

Linear List as C++ Abstract Class

```
template <class T>
class linearList {
public:
    virtual ~linearList() {};
    virtual bool empty() const = 0;
            //return true iff list is empty
    virtual int size() const = 0;
            //return the number of elements in list
    virtual T& get(int _index) const = 0;
            //return element whose index is _index
    virtual int indexOf(const T& _element) const = 0;
            //return the index of first occurrence of _element
    virtual void erase(int _index) = 0;
            //remove the element whose index is _index
    virtual void insert(int _index, const T& _element) = 0;
            //insert _element so that its index is _index
    virtual void output(ostream& out) const = 0;
```

Storing Lists

- Contiguous list (array)
  - Pros: easy to implement, excellent choice for static use.
  - Cons: time consuming for dynamic use, fragment may occur without carefully implementation.
- Linked list
  - Head pointer: Indicating the start.
  - NIL pointer (NULL pointer): Indicating the end.

Singly Linked Lists: Memory Layout

- Layout of $L = (a, b, c, d, e)$ using an array representation.

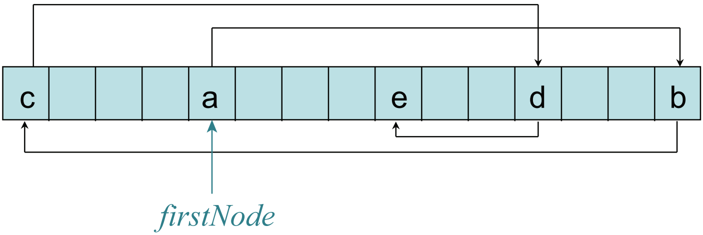
a	b	c	d	e										

- A linked representation uses an arbitrary layout.

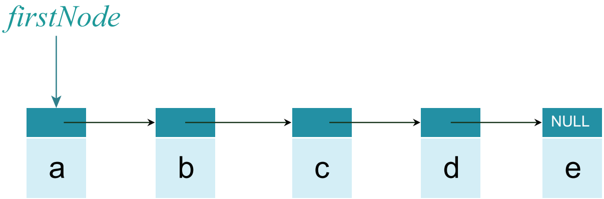
c				a				e			d			b

Linked Representation

- Use a variable *firstNode* to get to the first element *a*
- Pointer (or link) in *e* is *NULL*



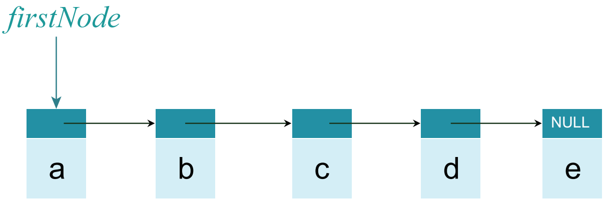
Normal Way To Draw a Linked List



- link or pointer field of node
- data field of node

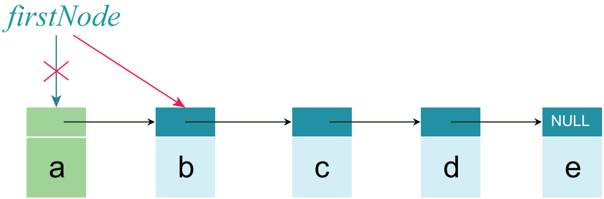
list::get(int _index)

- Start from the first node.
- get(2)
  - $desiredNode = firstNode \rightarrow next \rightarrow next$; // get to the 3^{rd} node
  - return $desiredNode \rightarrow element$;



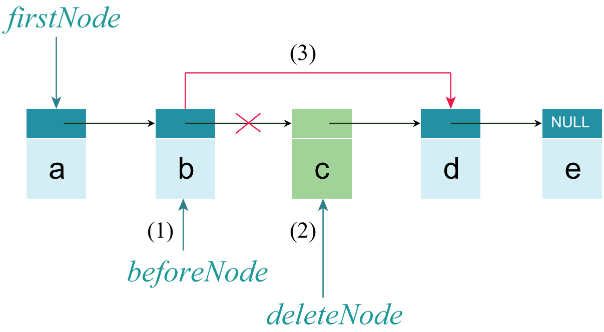
list::erase(0)

- Special case: need to change *firstNode*



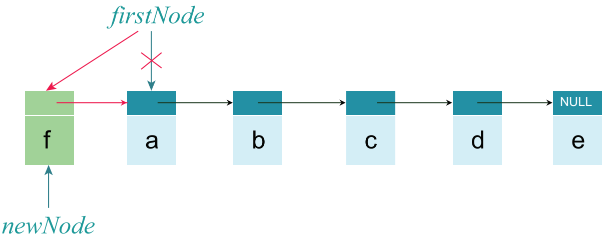
list::erase(2)

1. First get the *beforeNode*
2. Identify the *deleteNode*
3. Then change pointer in *beforeNode*



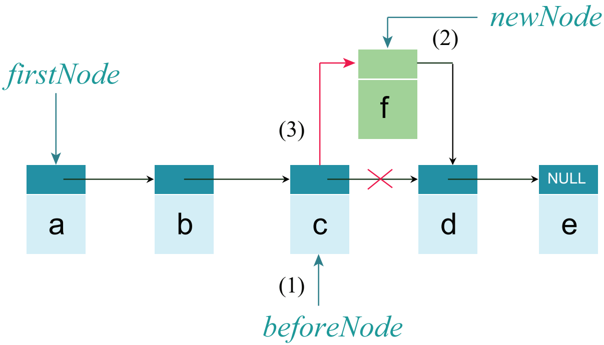
list::insert(0, ‘f’)

- Get a node, set its data and link fields
- Update *firstNode*



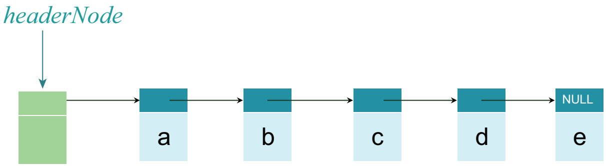
list::insert(3, 'f')

1. Find *beforeNode*
2. Create a node and set its data/link fields.
3. Link *beforeNode* to *newNode*

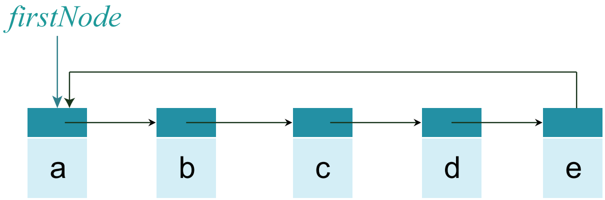


Variations

- List with a header node (dummy).

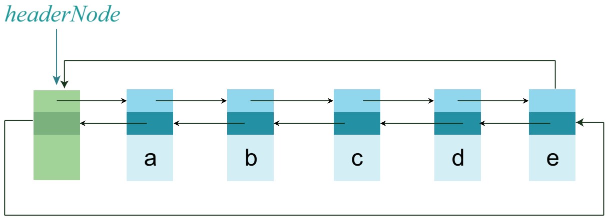


- Circular list



Doubly Linked Circular List with Header

- STL class std::list
  - Doubly linked circular list with header node.
  - Has many more methods than our list.



STL list

- #include <list>
- size()
- push_front(), push_back()
- pop_front(), pop_back()
- http://www.cplusplus.com/reference/stl/list/

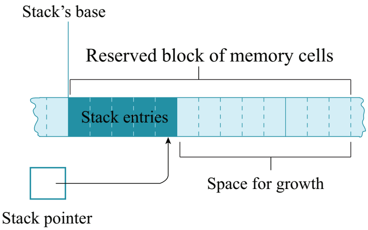
- iterator
  - Standard way to traverse a STL container

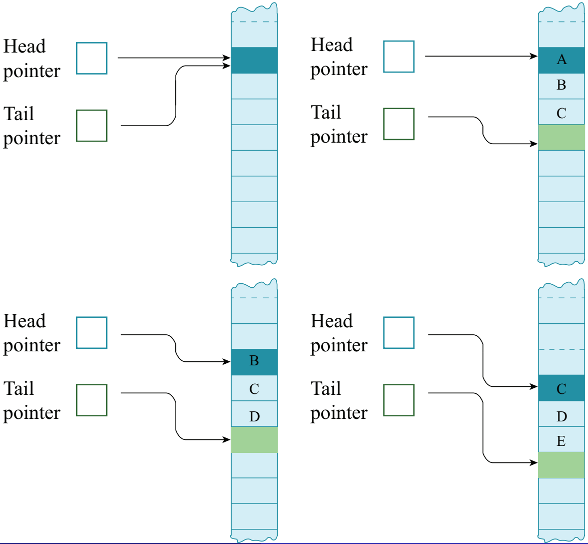
```
list<int> a;
    ....
for (list<int>::iterator it= a.begin(); it != a.end(); ++it)
    cout << *it << " ";
```

Stack & Queue Implementations

- Special cases of linked list
  - Stack: Recording the stack point
  - Queue: Recording head and tail
- Contiguous list
  - Stack: Array with a stack point
  - Circular queue

Stacks & Queues Implementations (contd.)





Abstract Stack Class

```
template <class T>
class stack {
public:
    virtual ~stack() {};
    virtual bool empty() const = 0;
        //return true iff stack is empty
    virtual int size() const = 0;
        //return the number of elements in stack
    virtual T& top() = 0;
        //return reference to the top element
    virtual void pop() = 0;
        //remove the top element
    virtual void push(const T& _element) = 0;
        //insert _element at the top of the stack
};
```

Derive from Array

- Stack top is either left end or right end.
- empty() → arrayList::empty() → $\Theta(1)$
- size() → arrayList::size() → $\Theta(1)$
- top() → get(0) or get(size() - 1) → $\Theta(1)$

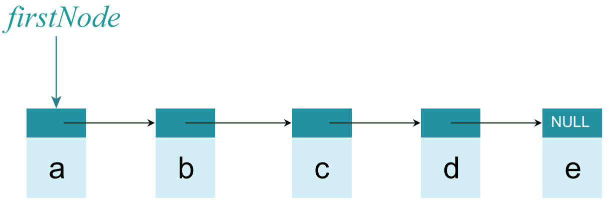
a	b	c	d	e									

Derive from Array (contd.)

- When top is left end
  - push(_element) $\rightarrow$ insert(0, _element) $\rightarrow$ $\Theta(n)$
  - pop() $\rightarrow$ erase(0) $\rightarrow$ $\Theta(n)$
- When top is right end
  - push(_element) $\rightarrow$ insert(size(), _element) $\rightarrow$ $\Theta(1)$
  - erase(size()-1) $\rightarrow$ $\Theta(1)$

Derive from Linked List

- Stack top is either left end or right end.
- empty() → list::empty() → $\Theta(1)$
- size() → list::size() → $\Theta(1)$



Derive from Linked List (contd.)

- When top is right end
  - top() $\rightarrow$ get(size()-1) $\rightarrow$ $\Theta(n)$
  - push(_element) $\rightarrow$ insert(size(), _element) $\rightarrow$ $\Theta(n)$
  - pop() $\rightarrow$ erase(size()-1) $\rightarrow$ $\Theta(n)$
- When top is left end
  - top() $\rightarrow$ get(0) $\rightarrow$ $\Theta(1)$
  - push(_element) $\rightarrow$ insert(0, _element) $\rightarrow$ $\Theta(1)$
  - pop() $\rightarrow$ erase(0) $\rightarrow$ $\Theta(1)$

Parentheses Matching

(	(	(	a	+	b	)	*	c	+	d	–	e	)	/	(	f	+	g	)	–	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

- Output pairs (u, v) such that the left parenthesis at position u is matched with the right parenthesis at v
  - (2,6), (1,13), (15,19), (21,25), (0,26)
- Also report missing parentheses
  - (a+b))*((c+d)
  - (0,4), right parenthesis at 5 has no matching left parenthesis, (8,12), left parenthesis at 7 has no matching right parenthesis

Parentheses Matching: Algorithm

- Scan expression from left to right
- When a left parenthesis is encountered, push its position to the stack
- When a right parenthesis is encountered, pop matching position from stack

Parentheses Matching: Example

(	(	(	a	+	b	)	*	c	+	d	−	e	)	/	(	f	+	g	)	−	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

2
1
0

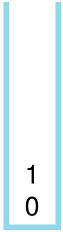
Actions

Push 0
Push 1
Push 2

Output

Parentheses Matching: Example (contd.)

(	**(**	**(**	**a**	**+**	**b**	**)**	*****	**c**	**+**	**d**	**−**	**e**	**)**	**/**	**(**	**f**	**+**	**g**	**)**	**−**	**(**	**h**	**+**	**j**	**)**	**)**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26



Actions

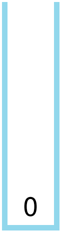
Push 0
Push 1
Push 2
Pop 2 → Output (2,6)

Output

(2,6)

Parentheses Matching: Example (contd.)

(	(	(	a	+	b	)	*	c	+	d	−	e	)	/	(	f	+	g	)	−	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26



Actions

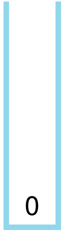
Push 0
Push 1
Push 2
Pop 2 → Output (2,6)
Pop 1 → Output (1,13)

Output

(2,6)
(1,13)

Parentheses Matching: Example (contd.)

(	(	(	a	+	b	)	*	c	+	d	–	e	)	/	(	f	+	g	)	–	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26



Actions

Push 0
Push 1
Push 2
Pop 2 → Output (2,6)
Pop 1 → Output (1,13)
Push 15
Pop 15 → Output (15,19)
Push 21
Pop 21 → Output (21,25)
Pop 0 → output (0,26)

Output

(2,6)
(1,13)
(15,19)
(21,25)
(0,26)

Parentheses Matching: Example (contd.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	1 0	1 1	1 2

Actions

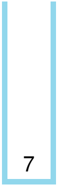
Push 0
Pop 0 → Output (0,4)
Pop → Empty stack!!!

Output

(0,4)
“right parenthesis at 5 has no matching left parenthesis”

Parentheses Matching: Example (contd.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12



Actions

Push 0
Pop 0 → Output (0,4)
Pop → Empty stack!!!
Push 7
Push 8
Pop 8 → Output (8,12)
7 still remains!!!

Output

(0,4)
"right parenthesis at 5 has no matching left parenthesis"
(8,12)
"left parenthesis at 7 has no matching right parenthesis"

Post-order Calculator

- $3 + 4 * 5 \rightarrow 3\ 4\ 5 * +$
- $(3 + 4) * 5 \rightarrow 3\ 4 + 5 *$

- Algorithm
  - For an operand token, push it into stack
  - For an operator token, pop tokens, operate, then push back the result.

Post-order Calculator: Example

3	4	+	5	*
Push 3	Push 4	Pop 4 Pop 3 Push 3+4	Push 5	Pop 5 Pop 7 Push 7*5
3	4 3	7	5 7	35

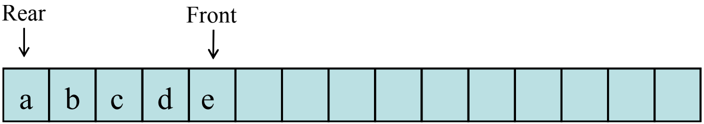
3	4	5	*	+
Push 3	Push 4	Push 5	Pop 5 Pop 4 Push 4*5	Pop 20 Pop 3 Push 3+20
3	4 3	5 4 3	20 3	23

Abstract Queue Class

```
template <class T>
class queue
{
 public:
   virtual ~queue() {};
   virtual bool empty() const = 0;
                    //return true iff queue is empty
   virtual int size() const = 0;
                    //return the number of elements in queue
   virtual T& front() = 0;
                    //return reference to the front element
   virtual T& back() = 0;
                    //return reference to the back element
   virtual void pop() = 0;
                    //remove the front element
   virtual void push(const T& _element) = 0;
                    //add _element at the back of the queue
}
```

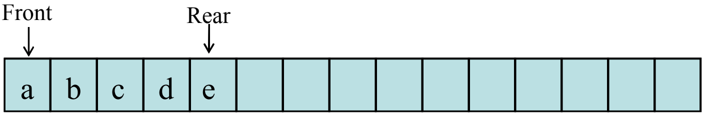
Derive from Array

- When front is right end & rear is left end
  - empty() $\rightarrow$ queue::empty() $\rightarrow$ $\Theta(1)$
  - size() $\rightarrow$ queue::size(0) $\rightarrow$ $\Theta(1)$
  - front() $\rightarrow$ get(size() - 1) $\rightarrow$ $\Theta(1)$
  - back() $\rightarrow$ get(0) $\rightarrow$ $\Theta(1)$
  - pop() $\rightarrow$ erase(size() - 1) $\rightarrow$ $\Theta(1)$
  - push(_element) $\rightarrow$ insert(0, _element) $\rightarrow$ $\Theta(n)$



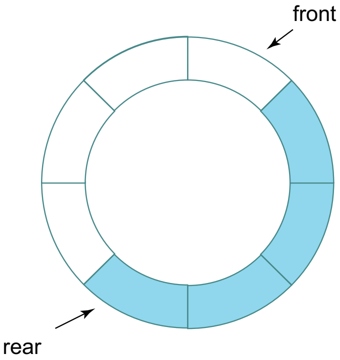
Derive from Array (contd.)

- When front is left end & rear is right end
  - empty() $\rightarrow$ queue::empty() $\rightarrow$ $\Theta(1)$
  - size() $\rightarrow$ queue::size(0) $\rightarrow$ $\Theta(1)$
  - front() $\rightarrow$ get(0) $\rightarrow$ $\Theta(1)$
  - back() $\rightarrow$ get(size()-1) $\rightarrow$ $\Theta(1)$
  - pop() $\rightarrow$ erase(0) $\rightarrow$ $\Theta(n)$
  - push(_element) $\rightarrow$ insert(size(), _element) $\rightarrow$ $\Theta(1)$



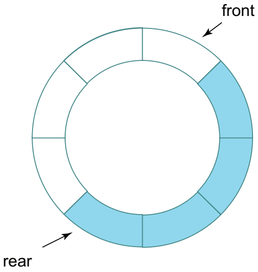
Can We Do Better?

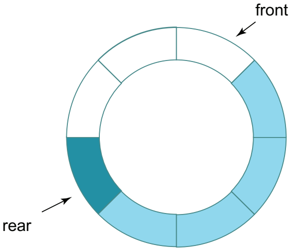
- To perform each operation in $\Theta(1)$ time (excluding array doubling), we need a customized array representation.
- Circular.



Push

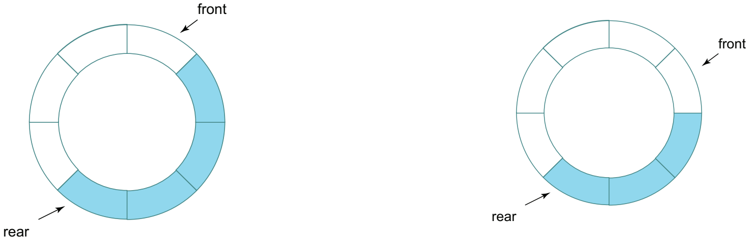
1. Move Rear clockwise. $rear = (rear + 1)\ \%\ arrayLength$
2. Then put into $queue[rear]$





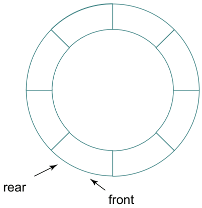
Pop

1. Move Front clockwise. $front = (front + 1)\ \%\ arrayLength$
2. Then erase $queue[front]$

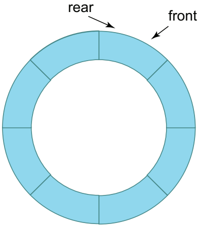


Empty & Full Queue

An empty queue.



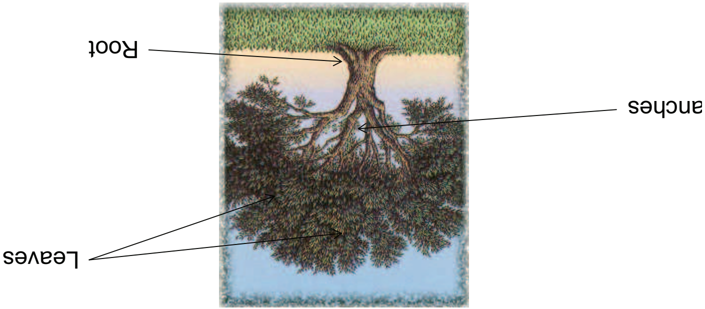
A full queue.



- Both *front* $==$ *rear*.
- Define an integer variable *size*.
  - Following each push do $++$*size*.
  - Following each pop do $--$*size*.
  - Queue is empty iff $(size == 0)$.
  - Queue is full iff $(size == arrayLength)$.

Nature Lover's View Of A Tree

Computer Scientist’s View



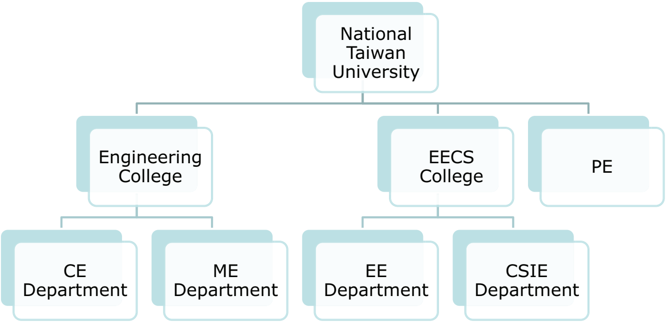
Linear Lists And Trees

- Linear lists are useful for serially ordered data.
  - $(e_0, e_1, e_2, ..., e_{n-1})$
  - Days of week.
  - Months in a year.
  - Students in this class.
- Trees are useful for hierarchically ordered data.
  - Employees of a corporation.
    - President, vice presidents, managers, and so on.

Hierarchical Data and Trees

- The element at the top of the hierarchy is the root.
- Elements next in the hierarchy are the children of the root.
- Elements next in the hierarchy are the grandchildren of the root, and so on.
- Elements that have no children are leaves.

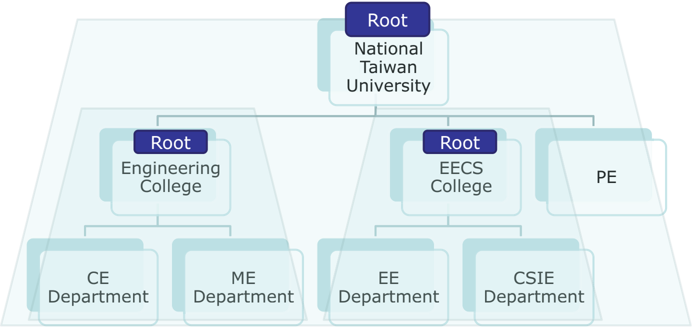
Example Tree



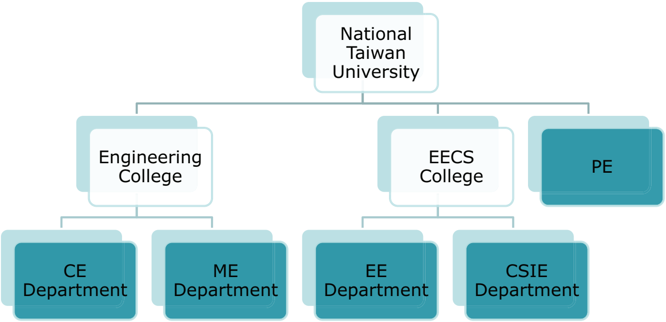
Definition

- Recursive definition.
- A tree t is a finite non-empty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t.

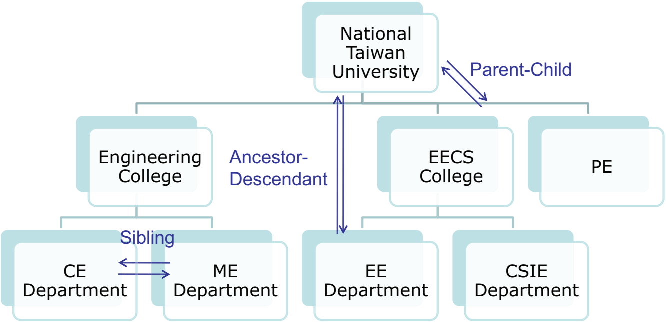
Example Tree



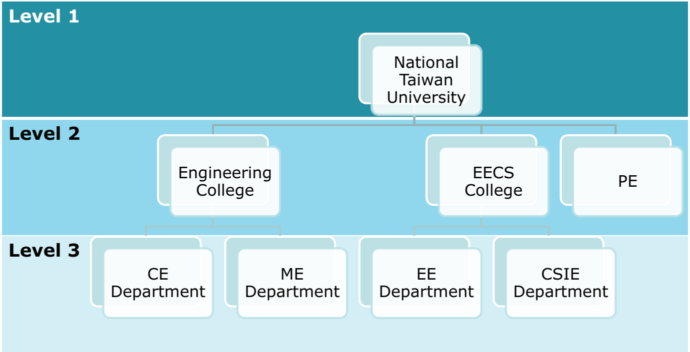
Leaves


National Taiwan University
Engineering College
EECS College
PE
CE Department
ME Department
EE Department
CSIE Department

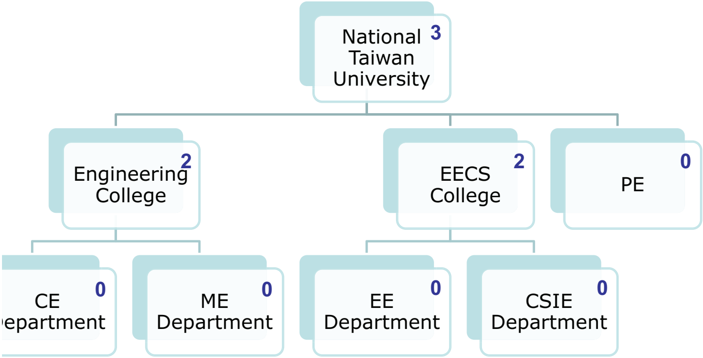
Parent, Children, Siblings, Ancestors, Descendants



Levels


Level 1
National Taiwan University
Level 2
Engineering College
EECS College
PE
Level 3
CE Department
ME Department
EE Department
CSIE Department

Node Degree = Number Of Children



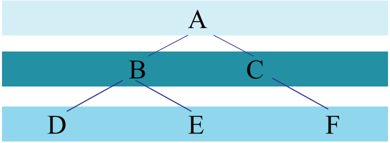
Tree Degree = Max Node Degree (=3)

Binary Trees

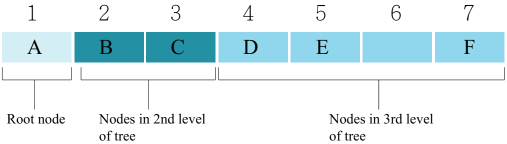
- Finite non-empty collection of elements.
- A binary tree has a root element.
- The remaining elements (if any) are partitioned into two binary trees.
- These are called the left and right subtrees.

Storing Binary Trees without Pointers

Conceptual tree

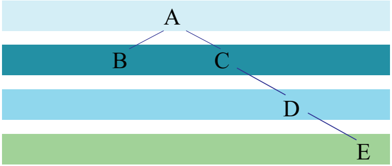


Actual storage organization

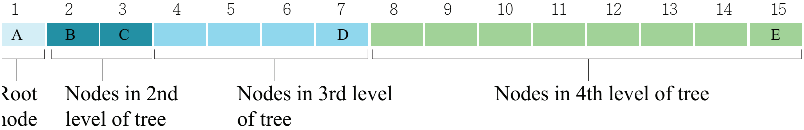


May Waste Lots of Memories...

Conceptual tree

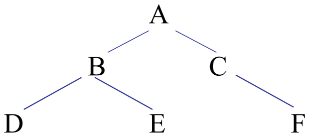


Actual storage organization

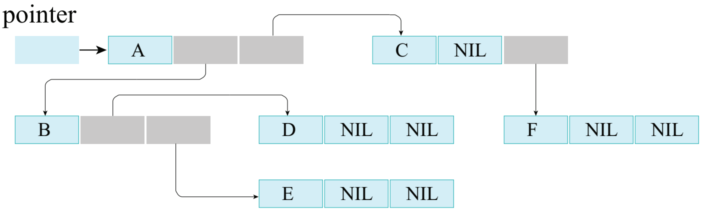


Storing Binary Trees with Pointers

Conceptual tree



Actual storage organization

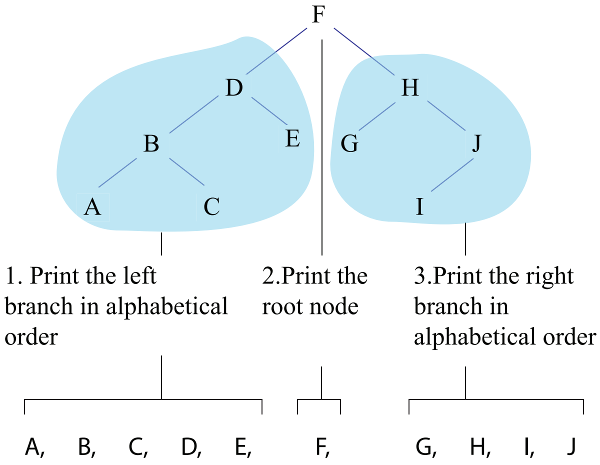


Definition of Binary Search Tree (BST)

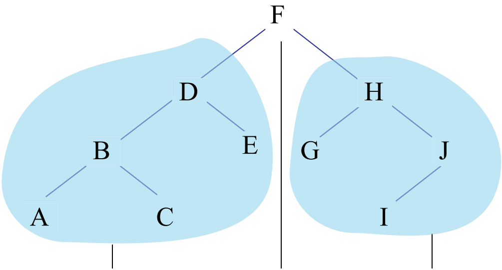
- A binary tree.
- Each node has a $(key, value)$ pair.
- For every node x, all keys in the left subtree of x are smaller than that in x
- For every node x, all keys in the right subtree of x are greater than that in x

- Operations
  - Traversal.
  - Search, insertion, deletion.
  - If the tree is balanced, insertion and search takes only $\Theta(\log n)$ of time, where n is the number of nodes.

Traverse in Order



Pre-Order and Post-Order



Pre-order: (1) root (2) left (3) right
F D B A C E H G J I

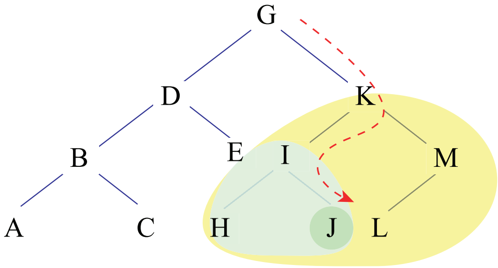
Post-order: (1) left (2) right (3) root
A C B E D G I J H F

Pre-Order and Post-Order

- In-order of a BST is always like sorting ascended.
- A BST is **uniquely decided** given its pre-order or post-order traversal (deciding the root and then splitting nodes into left and right).
- A binary tree is **uniquely decided** given its pre-order (or post-order) and in-order traversals.

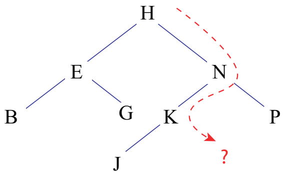
Search

- Very similar to binary search (may not be half-half).

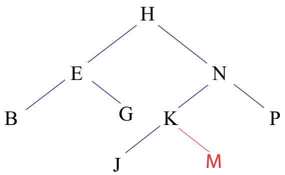


Insertion

A. Search for the new entry until its absence is detected.

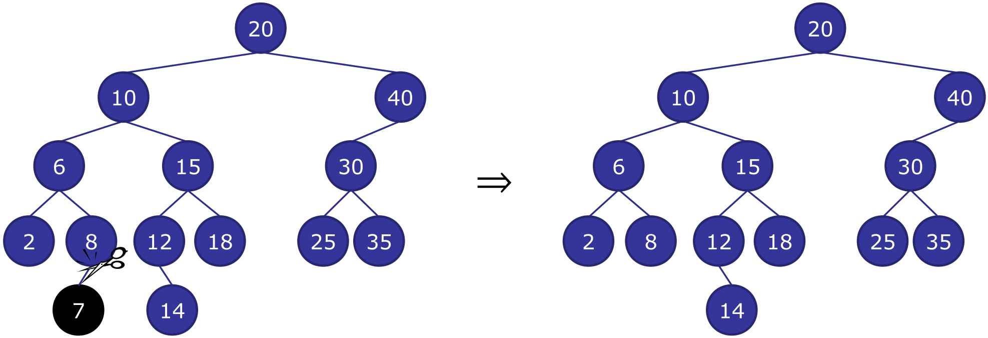


B. This is the position in which the new entry should be attached.



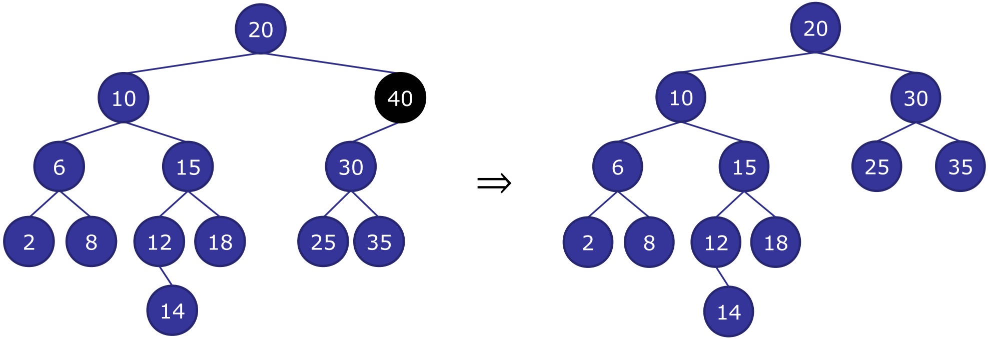
Delete A Leaf

- Erase a leaf element whose key is 7.



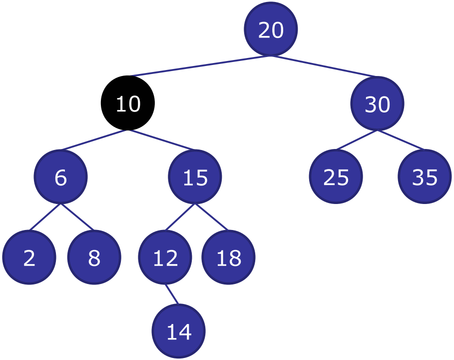
Delete A Degree-1 Node

- Erase a leaf element whose key is 40.



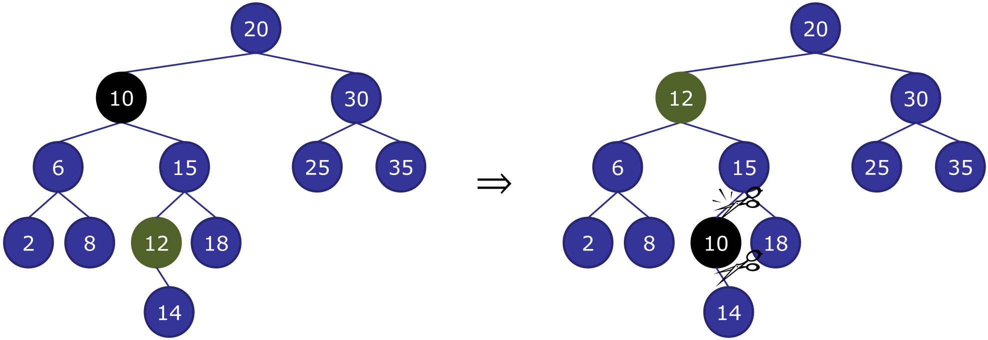
Delete A Degree-2 Node

- Erase an element whose key is 10.



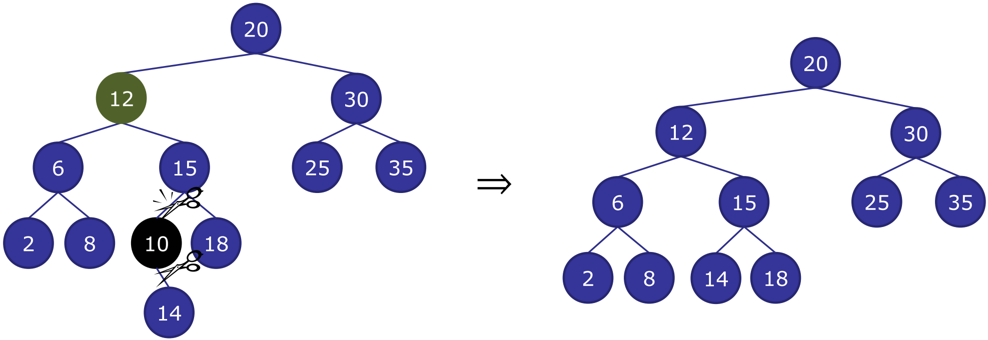
Delete A Degree-2 Node (contd.)

- Swap it with its successor (or predecessor).
  - The minimum node of the right subtree (keep going left).
  - Or the parent if itself is a left child.



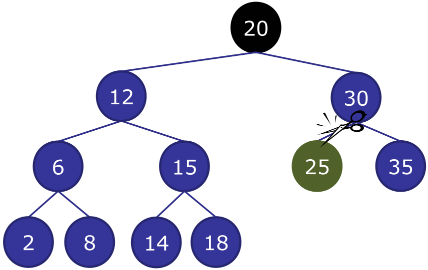
Delete A Degree-2 Node (contd.)

- Its successor has degree of 1 or 0.
- So simply cut and reconnect the rest of the tree.

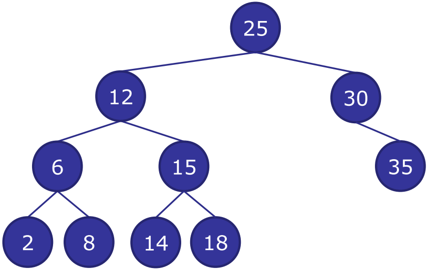


Delete A Degree-2 Node (contd.)

- Erase an element whose key is 20.

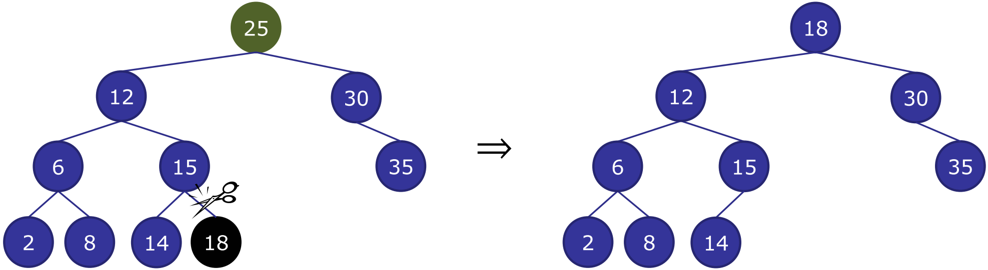


$\Rightarrow$



Delete A Degree-2 Node (contd.)

- Erase an element whose key is 18.



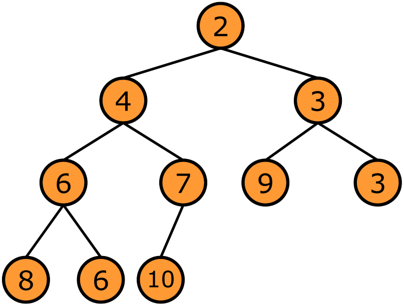
Priority Queue

- A stack pops the newest element.
- A queue pops the oldest element.
- What if we want to pop the most important element?
- If we associate elements with priorities, we'd like to pop the element with the highest priority.

- That data structure that accomplishes this task is called a priority queue.
- A binary heap is one method to implement a priority queue.

Binary Min Heap

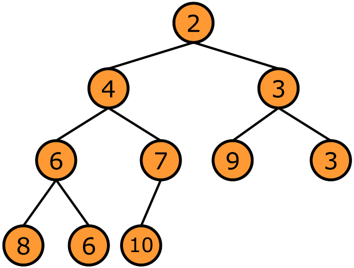
- A complete binary tree.
  - A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A min tree.
  - The key of each parent is no greater than any of its child.



- A min heap with 10 nodes.

Storing Binary Heap by Array

- The most common way to store a binary heap is by using an array.



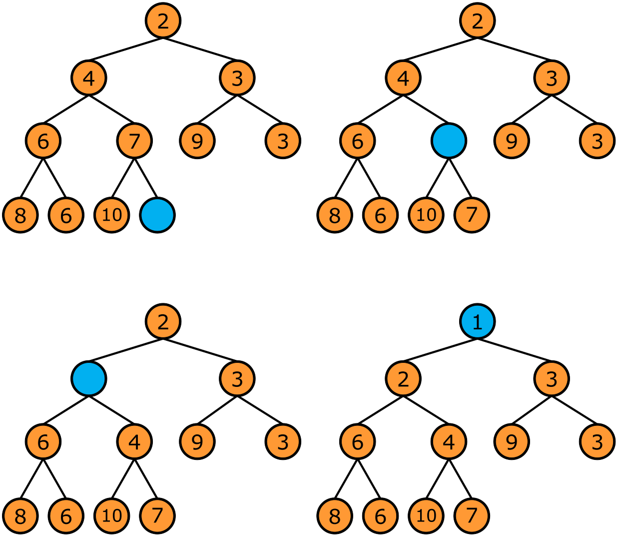
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$
2	4	3	6	7	9	3	8	6	10

- Traverse with index i:

Go to parent	Left child	Right child	Sibling
$(i-1)/2$	$2*i+1$	$2*i+2$	even: $i-1$, odd: $i+1$

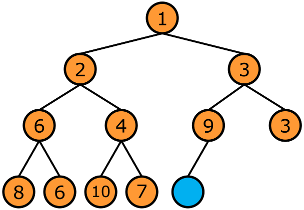
Push ‘1’

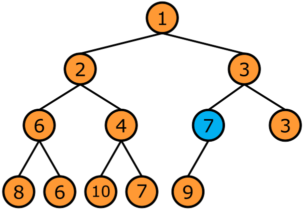
- The new element is always inserted as the last element.
- Then float up as needed.



Push ‘7’

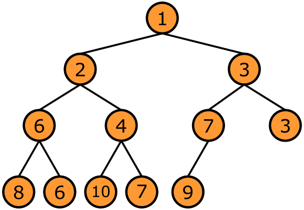
- The new element is always inserted as the last element.
- Then “float” up as needed.

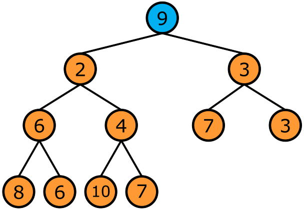




Pop

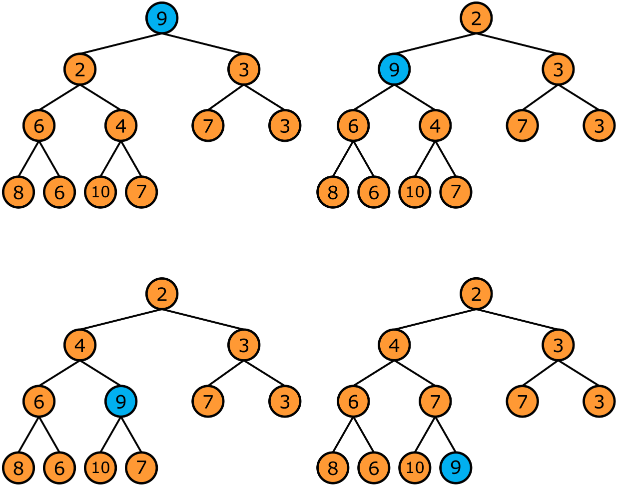
- Pop the root (smallest key).
- Replace it with the last element.
- Then "sink" down by choosing the "smaller" path.





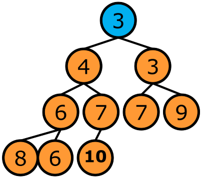
Pop (contd.)

- Pop the root (smallest key).
- Replace it with the last element.
- Then “sink” down by choosing the “smaller” path.



Pop Operation in Array

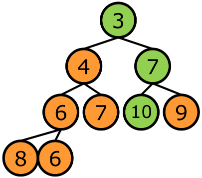
3	**4**	**3**	**6**	**7**	**7**	**9**	**8**	**6**	**10**
0	1	2	3	4	5	6	7	8	9



10	**4**	**3**	**6**	**7**	**7**	**9**	**8**	**6**	

3	**4**	**10**	**6**	**7**	**7**	**9**	**8**	**6**	
0	1	2	3	4	5	6	7	8	9

3	**4**	**7**	**6**	**7**	**10**	**9**	**8**	**6**	



Complexity of Heap

- For n elements, the height of the tree is $\Theta(\log n)$.
- Time complexity for both push and pop: $\Theta(\log n)$.
- We may accomplish sorting (called HeapSort) by keeping popping from a min heap: $\Theta(n \log n)$.
- We didn't show it, but modifying a key also costs $\Theta(\log n)$.