Advanced Programming Techniques

PART IV

Data Structures

Beniamin Bogosel

Ecole Polytechnique

Department of Applied Mathematics

Concept

- A data structure is a way of organizing and storing information: facilitate access or other purposes
- A data structure has an interface: a series of procedure to add, delete, access, re-organize the data
- a data structure can also store additional information $\star$ for example: the heap size for a heap
- An abstract data type (ADT) defines the properties of the structure and of the interface

In the following

- Dynamic sets: the number of elements may grow or decrease
- the objects may have multiple attributes, but we can concentrate only on the key with respect to which the object is identified
- Some data sets assume that there exists a total order on the keys: every two keys can be compared!

Standard operations for data structures

Two types: searching/access and modifications

Searching:

- $\textsc{Search}(S, k)$: returns a pointer x to an element in S such that $x.key = k$ or NONE if the element is not in S
- $\textsc{Minimum}(S)$, $\textsc{Maximum}(S)$: returns a pointer for the smallest/largest key
- $\textsc{Successor}(S, x)$, $\textsc{Predecessor}(S, x)$: returns a pointer for the immediatly greater/smaller than x in S, NONE if x is the maximum/minimum

Modifications:

- $\textsc{Insert}(S, x)$: insert the element x in S
- $\textsc{Delete}(S, x)$: delete the element x in S

Implementation of a data structure

- Generally, for an ADT multiple implementations are possible
- The complexity of operations depends on the implementation, not on the ADT
- Basic implementation bricks depend on the programming language
- A data structure can be implemented using another existing data structure

Standard data structures

- Pile: collection of objects – last in first out (LIFO)
- Queue: collection of objects – first in first out (FIFO)
- Double file: combine the two
- List: ordered collection of objects accessible by their position
- Vector: collection of objects accessible depending on their rank
- Tree: collection of objects structured like a tree
- Priority queue: access to the element of maximal key
- Dictionary: structure implementing the three operations: search, insert, delete

File

- Dynamic set of objects: LIFO (last in first out)
- Interface
  - $\textsc{Stack-Empty}(S)$: true if the pile is empty
  - $\textsc{Push}(S, x)$: pushes the value x on the pile S
  - $\textsc{Pop}(S)$: extracts and returns the value at the top of the pile

Applications:

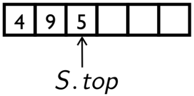
- The undo option
- calling functions in a compilator, etc

Implementation:

- with an array: fixed size *a priori*
- linked list (dynamically allocated)

Implementation through an array

- S is an array containing the elements of the pile
- $S.top$ is the position of the top of the pile




```
STACK-EMPTY(S)
    return S.top == 0
```

```
PUSH(S, x)
    if S.top == S.length
        error "overflow"
    S.top = S.top + 1
    S[S.top] = x
```

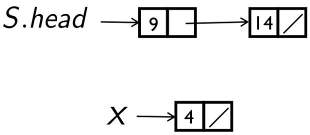
```
POP(S)
    if STACK-EMPTY(S)
        error "underflow"
    else S.top = S.top - 1
        return S[S.top + 1]
```

$\star$ Complexity in time and space $O(1)$: inconvenient – the space occupied does not depend on the number of objects

Recall: linked lists

- Data structure containing a sequence of elements containing
  - $x.data$: the data
  - $x.next$ pointer to the next element
  - (if doubly linked) $x.prev$ pointer to the previous element
- $L.head$: pointer to the first element in the list
- (if doubly linked) $L.tail$: pointer to the last element in the list
- void pointers at end of the list (in the corresponding direction)

⋆ S is a simple list




```
STACK-EMPTY(S)
  if S.head == NIL
        return TRUE
  else return FALSE
```

```
PUSH(S, x)
  x.next = S.head
  S.head = x
```

```
POP(S)
  if STACK-EMPTY(S)
        error "underflow"
  else x = S.head
        S.head = S.head.next
        return x
```

⋆ Complexity in time $O(1)$, complexity in space $O(n)$ for n operations

Application: paranthesis mismatch

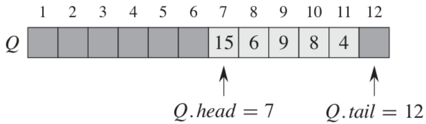
$\star$ Check the pairing of parantheses in a string of characters:
Example: $((x) + (y)]/2 \longrightarrow no$, $[-b + sqrt(4 * (a) * c)]/(2 * a) \longleftarrow yes$

$\star$ Solution based on a pile:

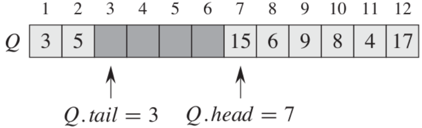
- Iterate through the characters of the string
- If we encounter a left paranthesis: push it on the pile
- If we encounter a right paranthesis pop the last element of the pile and check if it has the same type
- If a mismatch occurs return FALSE
- If the pile is empty at the end of the string, without any mismatches, return TRUE

Queue

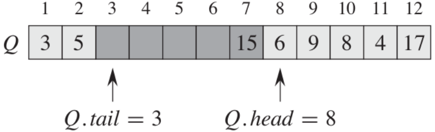
- data structure of the type FIFO (first in first out)
- Interface:
  - $\textsc{Enqueue}(Q, s)$: put the element s at the end of the queue Q
  - $\textsc{Dequeue}(S)$: pop the element at the top of the queue Q
- Implementation using a circular table
  - Q is an array of fixed length $Q.length$ (putting more than $Q.length$ elements in the file gives an error)
  - $Q.head$ is the position of the top of the queue
  - $Q.tail$ is the void position at the end of the queue
  - initially: $Q.head = Q.tail = 1$.



ENQUEUE(Q, 17), ENQUEUE(Q, 3), ENQUEUE(Q, 5)



DEQUEUE(Q) $\rightarrow$ 15



```
ENQUEUE(Q,x)
Q[Q.tail] = x
if Q.tail == Q.length
        Q.tail = 1
else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
        Q.head = 1
else Q.head = Q.head + 1
return x
```

- complexity in time $O(1)$, complexity in space $O(1)$

Using a list: straightforward implementation, space complexity depends on the length of the list

- Dynamic set of ordered objects based on their positions
- Generalizes the structures seen previously
- Interface:
  - functions for a double list (insertion and removal at beginning and end)
  - insert before or after a position
  - remove the element at a given position
  - replace an element
  - first and last elements of the list
  - previous or next elements in the list
- similarly to a double list: doubly linked list

Vector

- Dynamic set of objects occupying ranks given by consecutive integers
- Interface:
  - element at a given rank
  - replace at given rank
  - insert at given rank
  - remove at given rank
  - vector size

Implementation: list, extensible array

Complexity

- insertion
  - $O(n)$ for an individual operation: n is the size of the vector
  - $O(n)$ for n inserts at the end of the vector
  - $O(n^2)$ for n inserts at the beginning of the vector (shift all elements right...)
- removal: similar

Abstract data structure for a tree

- Main idea: data is associated to the nodes of a tree
- The nodes are accessible depending on their relative position in the tree

Interface: For a tree T and a node n

- $\textsc{Parent}(T, n)$: give the parent of the node n
- $\textsc{IsEmpty}(T)$: true if the tree is void
- $\textsc{Children}(T, n$: give a data structure containing the children of node n (ordered or not)
- $\textsc{IsRoot}(T, n)$: true if n is the root
- $\textsc{IsInternal}(T, n)$: true if the node is internal
- $\textsc{IsExternal}(T, n)$: true if the node is external
- $\textsc{GetData}(T, n)$: get data from node n
- $\textsc{Root}(T)$: give the root node
- $\textsc{Size}(T)$: the number of nodes in the tree

Examples of operations

- Depth of a node: count the number of parents till reaching the root node $O(n)$
- Compute the height of the tree
- etc...

Implementing a binary tree

First solution: level numerotation

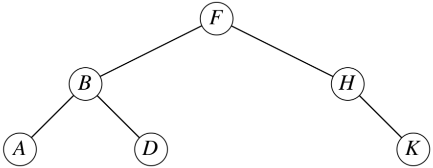
- ⋆ Root at position 1
- ⋆ from parent r reach children at positions $2r$ and $2r+1$
- ⋆ may contain void elements if the tree is not complete
- ⋆ complexity in space $O(2^n)$, complexity in time $O(1)$

Other options: linked structures - for each node keep pointers to its parent and its left/right children

Iterating through a binary tree

- A way of ordering the nodes in the tree, to iterate through them
- In depth
  - Infix (in order)
  - Prefix (in preorder)
  - Suffix (in postorder)
- In breadth

Infix traversal

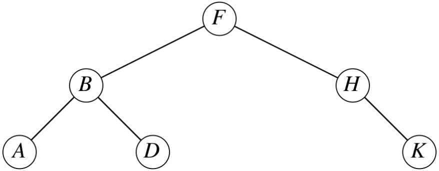


$$\Rightarrow \langle A, B, D, F, H, K \rangle$$

⋆ each node is visited after its left child and before its right child

```
INORDER-TREE-WALK(T, x)
if hasLeft(T, x)
        INORDER-TREE-WALK(T, LEFT(x))
print GETDATA(T, x)
if hasRight(T, x)
        INORDER-TREE-WALK(T, RIGHT(x))
```

Prefix traversal

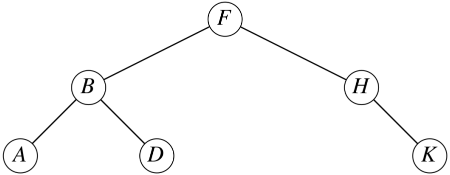


$$\Rightarrow \langle F, B, A, D, H, K \rangle$$

⋆ each node is visited before its children

```
PREORDER-TREE-WALK(T, x)
print GETDATA(T, x)
if HASLEFT(T, x)
    PREORDER-TREE-WALK(T, LEFT(x))
if HASRIGHT(T, x)
    PREORDER-TREE-WALK(T, RIGHT(x))
```

Postfix traversal



$$\Rightarrow \langle A, D, B, K, H, F \rangle$$

⋆ each node is visited after its children

```
POSTORDER-TREE-WALK(T, x)
if hasLeft(T, x)
        POSTORDER-TREE-WALK(T, LEFT(x))
if hasRight(T, x)
        POSTORDER-TREE-WALK(T, RIGHT(x))
print GETDATA(T, x)
```

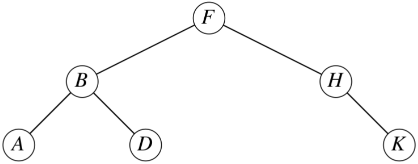
Complexity of traversals

All depth traversals are $\Theta(n)$ in time

- We need to pass through each node: $T(n) = \Omega(n)$
- By definition and recurrence of the traversals we find $T(n) = O(n)$

Breadth walks

- ⋆ visit the closest node to the root which was not yet visited
- ⋆ Using a queue in $\Theta(n)$



$$\Rightarrow \langle F, B, H, A, D, K \rangle$$

```
BREADTH-TREE-WALK(T)
Q =" Empty queue"
if not ISEMPTY(T)
    ENQUEUE(Q, ROOT(T))
while not QUEUE-EMPTY(Q)
    y = DEQUEUE(Q)
    print GETDATA(T, y)
    if HASLEFT(T, y)
        ENQUEUE(Q, LEFT(y))
    if HASRIGHT(T, y)
        ENQUEUE(Q, RIGHT(y))
```

Priority Queues

- Dynamical set of objects classified by a priority order
  - extract object with maximal priority
  - priority given by a key in a set having total order
- Interface:
  - $\textsc{Insert}(S, x)$: insert element x in S
  - $\textsc{Maximum}(S)$: return the element in S with largest key
  - $\textsc{Extract-Max}(S)$: delet the element of S having the largest key

Implementations

- Static array
  - Q is a static array with fixed length
  - Elements in Q are sorted in increasing order according to their keys
  - Complexity of extraction: $O(1)$
  - Complexity of insertion: $O(n)$
  - Space complexity: $O(1)$
- Linked list
  - Q is a linked list, sorted according to its keys.
  - Complexity of extraction: $O(1)$
  - Complexity of insertion: $O(n)$
  - Complexity in space: $O(n)$

Using a heap

- the top priority element is on top of the heap: $O(1)$
- insertion/extraction costs $O(\log n)$

```
HEAP-MAXIMUM(A)
    return A[1]
```

```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1) // reconstruit le tas
    return max
```

Dictionaries

Dictionaries are data dynamical data structures using comparable keys, supporting the following operations

- $\textsc{Search}(S, k)$ returns a pointer x towards an element in S such that $x.key = k$ and $NONE$ if k does not belong to S
- $\textsc{Search}(S, x)$ inserts the element x in the dictionary S. If an element with the same key exists, the value is replaced
- $\textsc{Delete}(S, x)$ removes the element x from S and does nothing if x is not in the dictionary.

We always assume that any two keys are comparable.

Dictionaries

Two general objectives:

- minimize the insert and access cost
- minimize the storage cost

Many implementations are possible.

Linked list

First solution:

- store the dictionary entries into a linked list
- to search an element, loop through the list
- Insertion (similarly for Deletion):
  - search the key
  - if found, then replace its value
  - if not found, then put it in the top of the list
- Complexity (worst case):
  - Insertion: $\Theta(N)$
  - Search: $\Theta(N)$
  - Delete: $\Theta(N)$

Second solution:

- store elements in a vector which we maintain sorted
- use binary search for searching: $\Theta(\log n)$
- Insertion: search the position and insert at given rank - shift all elements towards the right
- Deletion: delete and shift remaining elements towards the left
- Complexity:
  - Insertion $\Theta(N)$
  - Deletion $\Theta(N)$
  - Search $\Theta(\log N)$

Up to this point

	Worst case			Average		
Implementation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted vector	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Can we do better?

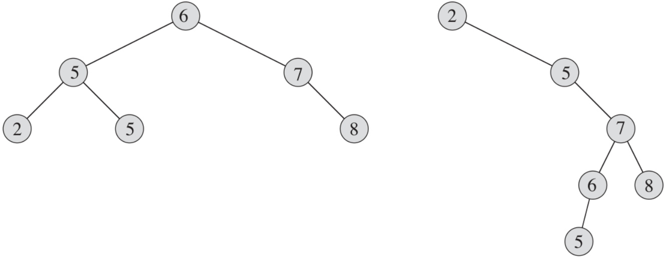
Remember: T is a tree, with a given root

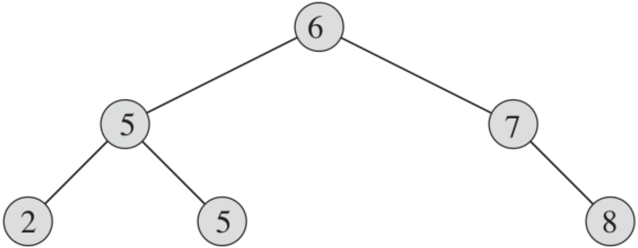
$\star$ for every node x we have access to

- $x.parent$: parent
- $x.key$
- $x.left$: left child
- $x.right$: right child

A binary search tree is a binary tree implementing a dictionary with operation costing $O(h)$ where h is the height of the tree!

- every node has an associated key
- the tree verifies the following property for two nodes x, y
  - If y is in the left sub-tree at x then $y.key < x.key$
  - If y is in the right sub-tree at x then $y.key \geq x.key$





$$\Rightarrow \langle 2, 5, 5, 6, 7, 8 \rangle$$

The infix walk in a binary tree allows to see the keys in increasing order

Searching in a binary tree

⋆ Binary search

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

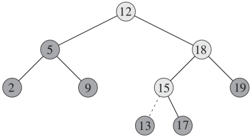
⋆ Complexity: $T(n) = O(h)$ where h is the height of the tree

⋆ Worst case: $h = n$

Other operations

- ⋆ Minimal/maximal key: leftmost-rightmost node, complexity $O(h)$
- ⋆ Successor, Predecessor: minimum/maximum in the right/left sub-trees, complexity $O(h)$
- ⋆ Insertion: search the key $x.key$ in the tree, insert x where the search stopped: $O(h)$

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y == NIL
    // Tree T was empty
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```



- ⋆ Deletion: $O(h)$

Binary search trees: summary

- ⋆ all operations are $O(h)$ where h is the height of the tree
- ⋆ inserting keys in arbitrary order may lead to different heights
- ⋆ The average height of a tree obtained is $\Theta(\log n)$

Summary

Implementation	Worst case Search	Insert	Delete	Average Search	Insert	Delete
List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted vector	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Binary search tree	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- ⋆ Can we get $O(\log n)$ in the worst case?
- ⋆ Yes: Weight-Balanced Trees: trees with guaranteed height of order $\Theta(\log n)$
- ⋆ Can we do better (on average) ? Yes: hash tables