Jan Maluszynski - HT 2006 3.1

**TTIT33 Algorithms and Optimization – DALG Lecture 3**

**Lecture 3 Algorithms**

**Basic ADT’s**

Stacks, Queues, Lists, Trees

Jan Maluszynski - HT 2006 3.2

**Content:**

- What is a *Data Structure*? [GT p.162, LD p.7, CL p.6]
- What is an *Abstract Data Type*? [GT p.60, LD p.15-17]
- ADT Stack [GT 5.1, LD Chapter 3, CL p.200-201]
- ADT Queue [GT 5.2-5.3, LD Chapter 3, CL p.202-203]
- ADT List; example of amortized analysis [GT 6.1-6.2, LD Chapter 3, CL 10.2, CL 17.1-4 (discusses in-depth amortized analysis)]
- ADT Tree [GT Chapter 7, LD 4.1-4.4, CL 10.4]

Jan Maluszynski - HT 2006 3.3

**Data structures**

[GT] p.162: “A data structure is a systematic way of organizing and accessing data.”

Two basic principles for organizing data:

- **Arrays (tables):**
  - data items in consecutive cells in the memory;
  - random access by indices
  
  ![Array indices](image)

- **Linked lists:**
  - data items scattered in the memory include pointers “connecting” them

Jan Maluszynski - HT 2006 3.4

**Abstract Data Types**

[GT] p.60: “An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of the parameters of the operations.”

Thus an ADT defines:

- A domain,
- Operations: what they do but not how

Implementation as a concrete data structure

Jan Maluszynski - HT 2006 3.5

**The Stack ADT**

The stack is a last-in-first-out memory

- Domain: arbitrary objects
- Operations:
  - push(object): inserts an element
  - pop(): removes and returns the last inserted element if the stack is empty
  - top(): returns the last inserted element without removing it, error if the stack is empty
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored
- Implemented as linked list or as array

Jan Maluszynski - HT 2006 3.6

**Some Applications of Stacks**

- Implementation of Recursive Procedures
- Evaluation of Arithmetic Expression
- Checking parentheses nesting (tag nesting in XML documents)
- History of visited pages in web browsers
- Undo operations in editors
- ….
### Implementation of Recursive procedures

Function `fact(int n):
if n = 0 then return 1
else f ← n ¤ fact(n-1); return f

- On recursive call: 
  - Push: arguments, local var, return addr (activation record)
  - Enter the call
- On return: 
  - Pop: the activation record
  - Go to the return addr

### Array-based stack

- Elements added from left
- Variable `t` shows the top

Algorithm `size()`
return `t + 1`

Algorithm `push(o)`
if `isEmpty()` then throw `FullStackException`
else
  `t ← t + 1`
  `S[t] ← o`
return `S[t + 1]`

Algorithm `pop()`
if `isEmpty()` then throw `EmptyStackException`
else
  `t ← t - 1`
return `S[t + 1]`

### The Queue ADT

The queue is a first-in first-out memory
- Domain: arbitrary objects
- Operations:
  - `enqueue(object)`: inserts an element at the end of the queue
  - `dequeue()`: removes and returns the first element of the queue, error if the queue is empty
  - `front()`: returns the last first element without removing it, error if the queue is empty
  - `size()`: returns the number of elements stored
  - `isEmpty()`: indicates whether no elements are stored
- Implemented as linked list or as circular buffer

### Circular buffer queue

- Use an array of size `N` in a circular fashion
- Two variables keep track of the front and rear
  - `f`: index of the front element (to be removed on dequeue)
  - `r`: index past the rear element: shows place for insertion
- Array location `r` is kept empty

### Queue Operations

#### Algorithm `enqueue(o)`
if `size() = N - 1` then throw `FullQueueException`
else
  `Q[r] ← o`
  `r ← (r + 1) mod N`

#### Algorithm `dequeue()`
if `isEmpty()` then throw `EmptyQueueException`
else
  `temp ← Q[f]`
  `f ← (f + 1) mod N`
return `temp`

### Queue Operations (cont.)

- Operation `enqueue` throws an exception if the array is full
- This exception is implementation-dependent

#### Algorithm `enqueue(o)`
if `size() = N - 1` then throw `FullQueueException`
else
  `Q[r] ← o`
  `r ← (r + 1) mod N`
List ADT

- The List ADT models a sequence of positions storing arbitrary objects.
- It establishes a before/after relation between positions.

Operations:
- size(), isEmpty()
- first(), last()
- prev(p), next(p)
- set(p, e)
- addBefore(p, e), addAfter(p, e)
- addFirst(e), addLast(e)
- remove(p)

Implementation: doubly-linked lists

Array List ADT

- The List ADT models a sequence of positions storing arbitrary objects.
- Unique reference to elements by an index 0,...,n-1.
- rank = index+1

Operations:
- size(), isEmpty()
- get(i)
- add(i, e)
- remove(i)

Implementation: array-based [GT]6.1.3

Trees: Basic terminology

A tree = set of nodes and edges, \( T = (V, E) \).

Nodes \( v \in V \) store data items in a parent-child relationship.

A parent-child relation between nodes \( u \) and \( v \) is shown as a directed edge \((u, v) \in E\), where the direction is from \( u \) to \( v \).

\( E \subseteq V \times V \)

Each node has at most one parent node; may have many siblings.

There is exactly one node that has no parent – the root node.

The degree of a node \( v \) is the number of children.

A node with 0 children is a leaf node or external node.

The other nodes are internal.

Example: \( ([a, b, c, d], ([a,b],[a,c],[c,d])) \)

ADT Tree

Operations on a single tree node \( v \) of a tree \( T \):
- parent(\( v \)) returns parent of \( v \), error if \( v \) root.
- children(\( v \)) returns set of children of \( v \).
- firstChild(\( v \)) returns first child of \( v \), or \( \Lambda \) if \( v \) leaf.
- rightSibling(\( v \)) returns right sibling of \( v \), or \( \Lambda \) if none.
- leftSibling(\( v \)) returns left sibling of \( v \), or \( \Lambda \) if none.
- isLeaf(\( v \)) returns true if \( v \) is a leaf (an external node).
- isInternal(\( v \)) returns true if \( v \) is a non-leaf node.
- isRoot(\( v \)) returns true if \( v \) is the root.
- depth(\( v \)) returns depth of \( v \) in \( T \).
- height(\( v \)) returns height of \( v \) in \( T \).

Special kinds of trees

Ordered tree: linear order among the children of each node.

Binary tree: ordered tree with degree \( \leq 2 \) for each node.

Empty binary tree (\( \Lambda \)): binary tree with no nodes.

Full binary tree: nonempty; degree is either 0 or 2 for each node.

Fact: number of leaves = 1 + number of interior nodes (proof by induction).

Perfect binary tree: all leaves have the same depth.

Fact: number of leaves = \( 2^h \) for a perfect binary tree of height \( h \) (proof by induction on \( h \)).

Complete binary tree: approximation to a perfect tree for \( 2^h \leq n < 2^{h+1} - 1 \).

Important and useful property!

ADT Tree (cont.)

Operations on entire tree \( T \):
- size() returns number of nodes of \( T \).
- root() returns root node of \( T \).
- height() returns the height of \( T \).

In addition for a binary tree:
- left(\( v \)) returns the left child of \( v \), or error.
- right(\( v \)) returns the right child of \( v \), or error.
- hasLeft(\( v \)) test if \( v \) has the left child.
- hasRight(\( v \)) test if \( v \) has the right child.
Tree representation: pointers

Type `Tnode` denotes a pointer to a structure storing node information:

```plaintext
record node_record
  nchilds: integer
  child: table<Tnode>[1..nchilds]
  info: infotype
```

- For binary trees: 2 pointers per node, `LC` and `RC`
- Alternatively, the pointers to a node’s children can be stored in a linked list.
- If required, a “backward” pointer to the parent node can be added.

Array representation of binary trees

Use a table<key,info>[0...n-1]

- `leftChild(i) = 2i + 1`
- `rightChild(i) = 2i + 2`
- `isLeaf(i) = i < n and 2i + 1 > n`
- `leftSibling(i) = i - 1`
- `rightSibling(i) = i + 1`
- `parent(i) = ⌊(i - 1)/2⌋`
- `isRoot(i) = (i = 0)`

Tree Traversals (1)

```plaintext
procedure preorder_visit(node v)
  do_something(v) {before any children}
  for all u ∈ children(v) do
    preorder_visit(u)

procedure postorder_visit(node v)
  for all u ∈ children(v) do
    postorder_visit(u)
  do_something(v) {after all children}

procedure inorder_visit(node v) {binary trees only!}
  inorder_visit(leftChild(v))
  do_something(v) {after all children}
  inorder_visit(rightChild(v))
```

Tree Traversals (2)

```plaintext
procedure level_order_visit(node v)
  Q ← mkEmptyQueue()
  enqueue(v, Q)
  while not isEmpty(Q) do
    v ← dequeue(Q)
    do_something(v)
    for all u ∈ children(v) do
      enqueue(v, Q)
```