DASI 2004

Tirgul 3

Topics of this Tirgul:

- Lists
- Vectors
- Stack
- Queue

DASI 2004

Abstract List Operations

- Create an empty list
- Test if the list is empty
- Provide access to elements at different positions in the list:
  - first
  - last
  - i-th element
- Insert a new element into the list
- Remove an element from the list
- Lookup an element by its contents
- Retrieve the contents of an element
- Replace the contents of an element
- Also useful: next(), previous()

DASI 2004

Linked List with no Pointers

- 2 linked lists in one array, one for the occupied cells and one for the free cells.
- Instead of using pointers to the next node, each cell holds the data + the index of the next node in the list.
- When adding an object a cell is removed form the free list and it's index is added to the occupied list.
- What is it good for ?
  - **Garbage collection.**
  - **A solution for a language with no pointers.**
    (and there are such languages!)

DASI 2004

Dynamic Arrays (Vectors)

- Many times you'll want to use arrays as for implementing advanced data structures.
- There is one problem though, the size of an array is predefined…
- You'll probably say, "My array is full? So what? Let's just create a bigger array and copy everything to it."
- But what about complexity?
- Looking at the cost of a single operation will not do, since it depends on the current state of the array.
- In order to evaluate our time efficiency we use "*Amortized Analysis*".

DASI 2004

Dynamic Arrays (Vectors)

- Let a be the initial size of array V
- Each time the array becomes full, we increase its size by c
- If the array is extended k times then $n = a + ck$
- The total number of basic operations is:
  $(a) + (a+c) + (a+2c) + \ldots + (a+(k-1)c) = a*k + c(1+2+\ldots+(k-1)) = a*k + c*k(k-1)/2 = O(k^2) = O(n^2)$
- We paid an amortized cost of n basic operations for each insert operation.

DASI 2004

Dynamic Arrays (Vectors)

- Let a be the initial size of array V
- This time, each time the array becomes full, we will double its size.
- If the array is extended k times then $n = a*2^k$
- The total number of basic operations is:
  $(a) + (2*a) + (2^2*a) + \ldots + (2^k*a) = \sum_{i=0}^{k} a*2^i = a*\sum_{i=0}^{k} 2^i = a*2^{k+1} - 1$
  $= O(k) = O(n)$
- We paid an amortized cost of one basic operation for each insert operation.
- But what happens if we allow the array to shrink as well?

Stack

- A collection of items that complies to a Last-In-First-Out (LIFO) policy:
- void push(Object o) - adds the object o to the collection.
- Object pop() - returns the most recently added object (and removes it from the collection).
- Object top() - returns the most recently added object (and leaves the stack unchanged).

Stack Implementation Using an Array

```
class StackA {
  private int maxSize;
  private int[] items;
  private int top;

  public StackA(int size) {
      maxSize = size;
      items = new int[maxSize];
      top = -1;
  }

  … // Stack operations
}
```

Stack Implementation Using an Array

```
public boolean isEmpty() {
  return (top == -1);
}

public void push(int item) {
  if (top >= maxSize-1) error(…);
  items[++top] = item;
}

public int pop() {
  if (isEmpty()) error(…);
  return items[top--];
}

public int top() {
  if (isEmpty()) error(…);
  return items[top];
}
```

java.util.Stack

- This java class implements a last-in-first-out stack of objects.
- The hierarchy: Object ⟶ Vector(cloneable) ⟶ Stack
- It has five methods.
  - empty()-checks if the stack is empty.
  - peek()-returns the top object without removing it.
  - pop()-pops…
  - push()-pushes…
  - search()-search for item in the stack.

Stack example: delimiter check

- Legal delimiters: {,[,(,),],}
- Each opening delimiter must have a matching closing one.

- Proper nesting:
  - a{bc[d]e}f(g) OK
  - a{bc[d}e]f(g) incorrect

- We can perform this task easily using a stack!

Stack example: delimiter check

```
// For all characters c of a string do:
switch (c) {
  case '(', '[', '{':
      stack.push(c);
      break;
  case ') ', '] ', '} ':
      if (stack.isEmpty())
            error(…);
      if (stack.pop() does not match c)
            error(…);
  default:
      break;
}

// When finished:
if (!stack.isEmpty())
  error(…);
```

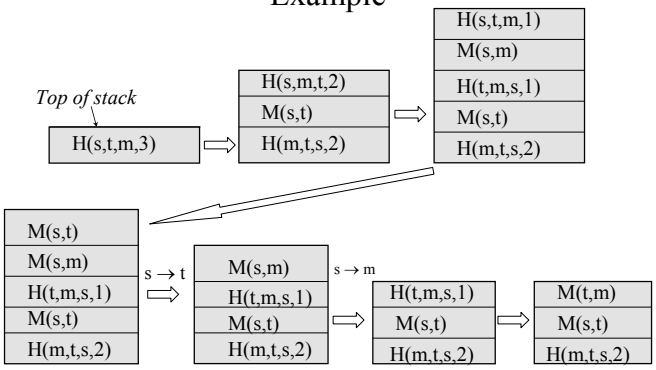
Using Stacks to Eliminate Recursion

- **Hanoi Tower, without recursion…**

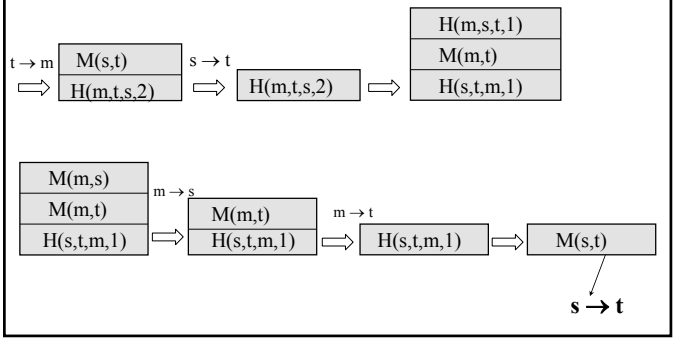
```
push( Hanoi('s', 't', 'm', k) )
while ( stack not empty )
  x = pop();
  if ( x is of type Hanoi )
    if ( x.discs > 1 )
        push( Hanoi(x.middle, x.to, x.from, x.discs-1) )
        push( Move(x.from, x.to) )
        push( Hanoi(x.from, x.middle, x.to, x.discs-1) )
    else
        push( Move(x.from, x.to) )
  else // x is of type Move
    print( x.from + " -> " + x.to);
```

```
Elements in stack are of
two types:
Hanoi(from, to,
      middle, discs)
Move(from, to)
```

Example



Example



Queue

- A collection of items that complies to a First-In-First-Out (FIFO) policy:
- void enqueue(Object o) - adds the object o to the collection.
- Object dequeue() - returns the least recently added object (and removes it from the collection).
- Object front() - returns the least recently added object (and leaves the queue unchanged). Also called peek().

Queue Implementation using a (circular) array

```
class QueueA {
  private int maxSize;
  private int[] items;
  private int front;
  private int back;
  private int numItems;


  public QueueA(int size) {
      maxSize = size;
      items = new int[maxSize];
      front = 0;
      back = maxSize-1;
      numItems = 0;
  }

  … // Queue operations
}
```

Queue Implementation using a (circular) array

```
public boolean isEmpty() {
  return (numItems == 0);
}

public boolean isFull() {
  return (numItems == maxSize);
}

public void enqueue(int item) {
  if (isFull()) error(…);
  back = (back+1) % maxSize;
  items[back] = item;
  numItems++;
}
```

Queue Implementation using a (circular) array

```
public int dequeue() {
   if (isEmpty()) error(…);
   int temp = items[front];
   front = (front+1) % maxSize;
   numItems--;
   return temp;
}

Public int peek() {
   if (isEmpty()) error(…);
   return items[front];
}
```

Question: can we do without keeping track of numItems?

The master theorem

Let $a \geq 1, b > 1$ be constants, let $f(n)$ be a function such that $n \geq 0 \Rightarrow f(n) \geq 0$ and let $T(n)$ be defined on non-negative integers by the recurrence:

$$T(n) = a*T(n/b) + f(n)$$

Then

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} * \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a*f(n/b) \leq c\, f(n)$ then $T(n) = \Theta(f(n))$

The master theorem (class version)

Let $a \geq 1, b > 1, c \geq 0$ be constants, and let $T(n)$ be defined on non-negative integers by the recurrence:

$$T(n) = aT(n/b) + n^c$$

Then

1. If $a/bc < 1$ ($c > \log_b a$) then $T(n) = \Theta(n^c)$
2. If $a/bc = 1$ ($c = \log_b a$) then $T(n) = \Theta(n^c \log_b n)$
3. If $a/bc > 1$ ($c < \log_b a$) then $T(n) = \Theta(\log_b a)$

The master theorem

T(n) = a*T(n/b) + f(n)		***T(n) = aT(n/b) + n^c***
1. If $f(n) = O(n^{\log_b a - \varepsilon})$ then $T(n) = \Theta(n^{\log_b a})$	⇨	1. If $a/bc < 1$ ($c > \log_b a$) then $T(n) = \Theta(n^c)$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} * \log n)$	⇨	2. If $a/bc = 1$ ($c = \log_b a$) then $T(n) = \Theta(n^c \log_b n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a*f(n/b) \leq c\, f(n)$ then $T(n) = \Theta(f(n))$	⇨	3. If $a/bc > 1$ ($c < \log_b a$) then $T(n) = \Theta(\log_b a)$