Parma University, Math. Department

Algorithms and Data Structures (Elementary Data Structures)

prof. Marco Pellegrini

Spring 2002, Lecture 3

Summary of lecture 3

- Abstract data types vs. data types
- Stacks, queues
  array implementation,
  list implementation,
  array simulating lists
  rooted trees

Data Structures

By "Data Structure" we indicate an organization of the "state" of the memory (Tape in a TM, Registers in a RAM) that speeds up the execution of algorithms.

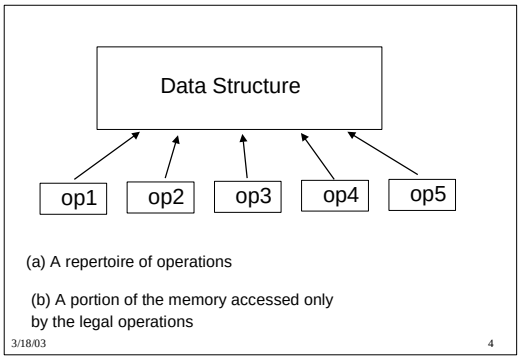
Example: Storing an increasing sorted sequence of n distinct numbers in *consecutive* registers R[x],R[x+1],…,R[x+n],

gives us structural information:

- the median element is in register $R[x+\lfloor n/2 \rfloor]$.

- alle elements in $R[x],\ldots,R[x+\lfloor n/2 \rfloor-1]$ are smaller than the median.

Abstract Data Structure



(a) A repertoire of operations

(b) A portion of the memory accessed only by the legal operations

Abstract Data Structures--Data Structures

(a) A sequence of operations is performed on the data at different momets during the execution of a *master* algorithm.

(b) The outcome of an operation depends on the state of the Data Structure and on the input parameters of the op .

(c) The state of the Data Structure does not change between operations.

(d) Only Operations can change the state of the data structure.

(e) In an Abstract Data Structure we specify only the operations (what) not the implementation of the operations (how).

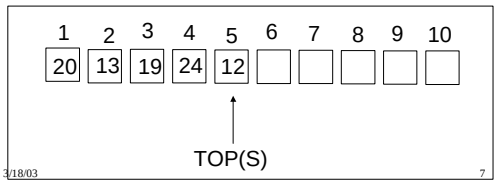
(f) In a Data Structure we specify also how the memory is organized and how the operations are performed.

The ADS Stack

- Stacks are ADS in which we have 2 main operations available:
  1) INSERT an element,
  2) GET-and-Delete the element that has been in the stack for the *shortest time*.
- This "policy" is called LIFO (last-in-first-out).
- Used in Convex Hull algorithm (in Geometry),
  Run-time execution support (O.S.).
- Insert is usually called PUSH,
  get-and-delete is usually called POP.

Implementation of Stacks using Arrays

- We use an array A[1..N] to hold a Stack S of at most N elements.
- The variable TOP(S) hold the index of the last element inserted. The elements of S are in the positions A[1...TOP(S)].



Operations on Stacks.

(Empty?, PUSH, POP)

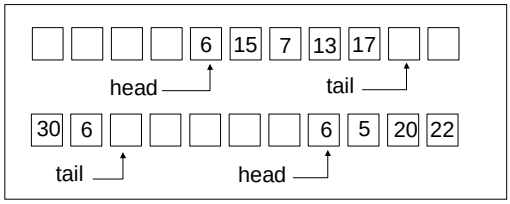
- EMPTY?(S: stack): boolean
  IF TOP(S)=0 THEN TRUE
  ELSE FALSE
  END IF
- PUSH(S: stack,x: element)
  TOP(S) := TOP(S)+1
  IF TOP(S) > N THEN Error
  S[TOP(S)] := x
- POP(S: stack): element
  IF EMPTY?(S) THEN Error
  ELSE TOP(S) := TOP(S)-1
  RETURN S[TOP(S)+1]
  END IF.

The ADS Queue

- Queues are data structures holding data with two operations:
  1) Insert an element,
  2) Get-and-delete the element that has been in the queue for the *longest time*.
- This will be used for visiting graphs in (Breadth First Search).
  It is used in queuing systems (client/server).
- The insert operation is usually called ENQUEUE, the get-and-delete is usually called DEQUEUE.
- The "policy" of Queues is called FIFO (First-in-first-out).

Implementation of Queues using Arrays.

- A Queue is resident in an array A[1..N]. The head of the queue Q is denoted by HEAD(Q). The tail of the queue Q is denoted by TAIL(Q).
- The elements of the queue Q are in positions HEAD(Q), HEAD(Q)+1,...TAIL(Q)-1.



Operations on Queues

(enqueue and dequeue)

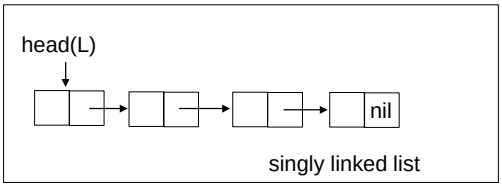
- When HEAD(Q)=TAIL(Q), then Q is empty.
  When HEAD(Q)=TAIL(Q)+1, then Q is full.
- ENQUEUE(Q: queue,x: element) % without error checks
  Q[TAIL(Q)] := x
  IF TAIL[Q]= N
  THEN TAIL(Q) := 1
  ELSE TAIL(Q) := TAIL(Q)+1
  ENDIF
- DEQUEUE(Q:queue);element % without error checks
  x := Q[HEAD(Q)]
  IF HEAD(Q) = N
  THEN HEAD(Q) := 1
  ELSE HEAD(Q) := HEAD(Q) +1
  ENDIF
  RETURN x

Exercises

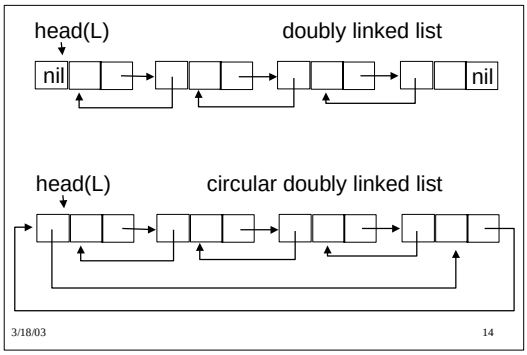
- Put error checks in the procedures ENQUEUE and DEQUEUE. (That is, detect underflow and overflow situations). (CLR 11.1-4).
- HW1.1 : (CLR 11.1-5) :
  write operations to insert and delete at both ends of a queue (also called a deque).

Linked lists implementation.

- We saw the ADS Stack and Queue plus their implementation based on arrays.
- We shall see briefly alternative implementations of the same ADS using linked lists.
- Linked lists are based on pointers.



Several types of lists



Some operations on lists

- Type element = record (key: number, next: ^element, prev: ^element)
- head(L): ^element
-
```
List-seach(L: list ,k:number)
 x := head(L)
 while x ≠NIL and key[x] <> k DO
    x := next[x]
 end while
 return x
```
- List-search tests whether k is in the list, it returns NIL if it is not there or the pointer to the element containing k.
- If L has n elements what is the worst case time function for List-search? The best? the average?

Other operations on doubly linked lists

- Insert an element x at the beginning of L
-
```
List-insert(L: list , x:^element(k,NIL,NIL))
 next[x]:= head[L]

 IF head(L) ≠ NIL
  THEN prev[head[L]]:= x
 END IF
 head[L]:= x
 prev[x] := NIL
```
- Note that this operation only changes the state of the list, there is no output returned.
- What is the time cost of one List-insert?

Deletion from doubly-linked lists

-
```
List-Delete(L: list ,x:^element)
% x points to an element in L
IF prev(x) ≠ NIL
  THEN next(prev(x)) := next(x)
  ELSE head(L) := next(x)
END IF
IF next(x) ≠ NIL
  THEN prev(next(x)) := prev(x)
```
- What is the time to delete a element from L, given a pointer to it?
- Exercise: use Insert-list and delete-list to implement PUSH and POP on a stack.

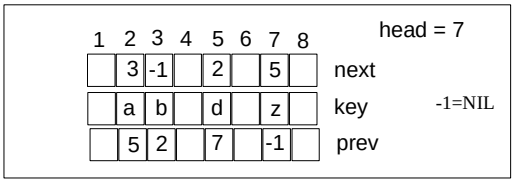
Tips and exercises.

- When writing code on pointer structures it is always a good idea to have a little drawing on a side.
- Sometimes it is useful to place a dummy element (sentinel) at the beginning or end of lists to simplify testing when we reach the beginning or the end of the list. (CLR page 207).
- HW1.2 (CLR 11.2-5).

Write code to implement the disjoint-set-union operation using lists such that it takes time O(1).

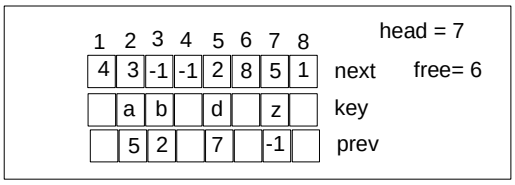
Lists and array

- We saw that the ADS Stack can be implemented using arrays or list. Similarly for the ADS Queue.
- Now we take it a step further using arrays to simulate list.
- This is useful when (1) The language does not provide pointers or (2) when you want to control tightly the use of storage.



Allocation and Garbage collection

- In Pascal, in C to create objects we use the primitives new, alloc, malloc. To reclaim unused memory there are garbage collectors.
- We have to simulate them, by keeping all the free slots in a separate list (headed by FREE)
- The free list (singly connected) works as a Stack

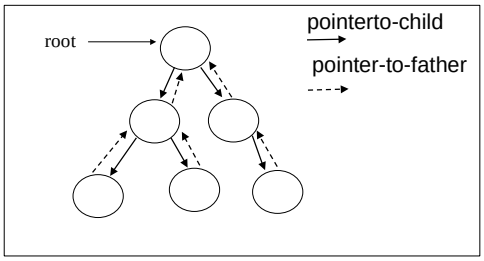


Exercises

- Exercise: Write a code for the operations free(x) that takes an (unused) index x and put it in the free-list.
- Exercise: Write a code for Allocate-object() which which returns a an index of a free location.

Trees

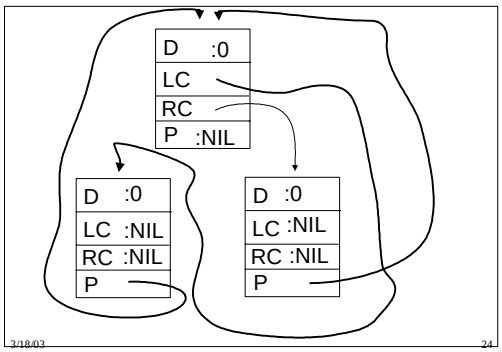
Using pointers we can create other structure besides lists: trees, graphs, etc.
Here we see the of a rooted binary tree.



Example in Pascal

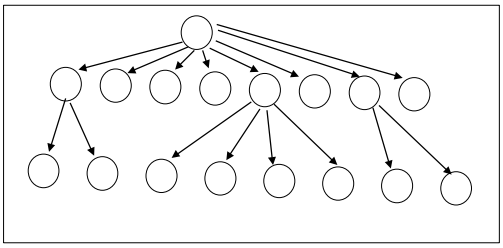
- TYPE node =
  RECORD
  data = Data-type
  left-child = ^node
  right-child = ^node
  parent = ^node
  END RECORD
- Then in the program we can build a tree dynamically:
  T := new(node)
  T1 := new(node)
  T2 := new(node)
  T1^.parent := T
  T2^.parent := T
  T^.left-child := T1
  T^.right-child := T2

Result of the execution of the code.

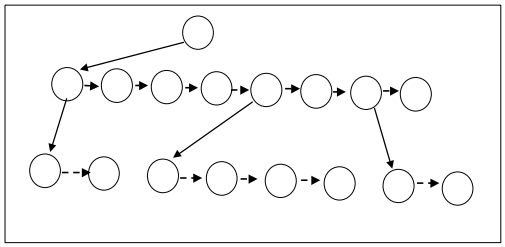


n-ary trees

- Sometimes we deal with where nodes can have many children, and sometimes a variable number of them.
- In this case we can use the left-child/right-sibling representation.



- In the left-child/right-sibling representation we use always only 2 pointers per node.



Conclusions

- Memory structure + operations form a Data Structure (DS).
- If we are interested only in what the operations do and not how it is done we have an Abstract Data Structure (ADS) also called Abstract Data Type (ADT)
- We can have different DS for the same ADS, for example one based on an array-implementation, one on a list implementation.
- For the same ADS we can analyze the time of the several corresponding DS to determine the best for the task at hand.