Applied Programming and Computer Science, DD2325/appcs14

PODF, Programmering och datalogi för fysiker, DA7011

Lecture 5, C programming: Abstract Data Types

A. Maki, C. Edlund
December 2014

Abstract Data Types (ADTs)

An abstract data type (ADT) is a mathematical model for:

- a certain class of data structures that have similar behavior, or
- certain data types that have similar semantics.

Examples:

- Stack (LIFO: Last-In, First-Out)
- Queue (FIFO: First-In, First-Out)
- Tree

Stack

- push
  - precond: The stack has been created and is not full.
  - postcond: The element has been stored as the stack's top element. The updated stack is returned.
- pop
  - precond: The stack has been created and is not empty.
  - postcond: The top element of the stack has been removed and is returned. The updated stack is returned as well.
- top
  - precond: The stack has been created and is not empty.
  - postcond: A copy of the top element of the stack is returned.

Stack

```
#include <stdio.h>
#include <stdlib.h>


struct listNode
{
  char data;
  struct listNode *nextPtr;
};


typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;


int isEmpty(ListNodePtr sPtr)
{
  return sPtr == NULL;
}
```

Stack (cont.)

```
void push(ListNodePtr *sPtr, char value)
{
  ListNodePtr newPtr;

  newPtr=(ListNode *) malloc(sizeof(ListNode));

  if (newPtr != NULL){
    newPtr->data = value;
    newPtr->nextPtr = *sPtr;
    *sPtr = newPtr;
  }
  else
      printf("Out of memory!! \n\n");
}
```

Stack (cont.)

```
char pop(ListNodePtr *sPtr)
{
  ListNodePtr currPtr;
  char value;

  currPtr = *sPtr;
  value = currPtr->data;
  *sPtr = currPtr->nextPtr;
  free(currPtr);
  return value;
}
```

Stack (cont.)

```
void delete (ListNodePtr *sPtr)
{
  char value;

  if (isEmpty(sPtr))
    {printf("The stack is empty! \n\n");}
  else
    {
      value = pop(sPtr);
      printf("Popping :%c \n", value);
    }
}
```

```
main()
{
  ListNodePtr startPtr = NULL;
  char item;
  int noOfNodes = 0;

  printf("Write initial data: ");
  scanf("\n%c",&item);

  while (item != 'q'){
      if (item != 'p'){
        push(&startPtr, item);
        noOfNodes ++; }
      else{
        delete(&startPtr);
        noOfNodes --; }
      printf("Write data: ");
      scanf("\n%c",&item);
    }
  printf("%d\n", noOfNodes);
}
```

Queue

A list with the restriction that insertion can be performed at one end (rear) and delection can be at the other end (front).

Seen in

- Real world
  - people/taxies waiting in a line/queue
  - cars on an assembly line
- Operating systems
  - print queue
  - queue of processes to be run

Queue

- FIFO: Retrieves elements in the order they were added
  - Elements are stored in order of insertion.
  - Can add to the end of the queue, and only remove the front of the queue.
- Queue operations
  - enqueue: Add an element to the back.
  - dequeue: Remove the front element.
  - isEmpty

Queue

```
#include <stdio.h>
#include <stdlib.h>

struct listNode {
  char data;
  struct listNode *nextPtr;
};

struct queueNode {
  struct listNode *headPtr, *tailPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

typedef struct queueNode queueNode;
typedef queueNode *queueNodePtr;

int isEmpty(queueNodePtr qPtr){
  return qPtr->headPtr == NULL;}
```

Queue (cont.)

```
void createQueue(queueNodePtr *qPtr)
{
  queueNodePtr newPtr;

  newPtr=(queueNode *) malloc(sizeof(queueNode));

  if (newPtr != NULL){
    newPtr->headPtr = NULL;
    newPtr->tailPtr = NULL;
    *qPtr = newPtr;}
  else
    printf("Out of memory!! \n\n");
}
```

Queue (cont.)

```
void enqueue(queueNodePtr qPtr, char value)
{
  ListNodePtr newPtr;
  newPtr=(ListNode *) malloc(sizeof(ListNode));

  if (newPtr != NULL){
    newPtr->data = value;
    newPtr->nextPtr = NULL;

    if (isEmpty(qPtr)){
      qPtr->headPtr = newPtr;
      qPtr->tailPtr = newPtr;}
    else{
      (qPtr->tailPtr)->nextPtr = newPtr;
      qPtr->tailPtr = newPtr;}
  }
  else
    printf("Out of memory!! \n\n");
}
```

Queue (cont.)

```
char dequeue(queueNodePtr qPtr)
{
  ListNodePtr currPtr;
  char value;

  currPtr = qPtr->headPtr;
  value = currPtr->data;
  qPtr->headPtr = currPtr->nextPtr;
  free(currPtr);
  return value;
}
```

How to traverse a tree?

Algorithm for preorder:

```
If the tree is not empty
   print current data on screen
   traverse the left subtree preorder
   traverse the right subtree preorder
```

How to traverse a tree? (cont.)

Algorithm for inorder:

```
If the tree is not empty
   traverse the left subtree inorder
   print current data on screen
   traverse the right subtree inorder
```

Tree

```
#include <stdio.h>
#include <stdlib.h>


struct treeNode
{
  char data;
  struct treeNode *leftPtr, *rightPtr;
};


typedef struct treeNode treeNode;
typedef treeNode *treeNodePtr;


int isEmpty(treeNodePtr tPtr)
{
  return tPtr == NULL;
}
```

Tree (cont.)

```
void insert(treeNodePtr *tPtr, char value)
{
  treeNodePtr newPtr;

  if (isEmpty(*tPtr)){
    newPtr=(treeNode *) malloc(sizeof(treeNode));
    if (newPtr != NULL){
      newPtr->data = value;
      newPtr->leftPtr = NULL;
      newPtr->rightPtr = NULL;
      *tPtr = newPtr;}
    else
      printf("Out of memory!! \n\n");}
  else{
    if ((*tPtr)->data > value)
      {insert(&(*tPtr)->leftPtr, value);}
    else
      {insert(&(*tPtr)->rightPtr, value);}}
}
```