Stacks and Queues. Dynamic Memory Allocation.

Lecture 5

2015

Stacks, queues and dequeues

- Stacks and queues are dynamic sets such that the element removed is pre-specified.
- In a *stack*, the element removed is the last element inserted. So a stack implements the principle *Last In First Out* – LIFO.
- In a *queue*, the element removed is the first element inserted. So a queue implements the principle *First In First Out* – FIFO.
- A *double ended queue – dequeue* is a list such that insertions and removals can be explicitly made to either ends of the list.
- Stacks & queues are implemented using arrays or linked lists.

2015

Applications of stacks

- Stacks are used in:
  - Compilers of programming languages:
    - For language parsing
    - For code generation
  - Language implementation:
    - To implement procedure calls
  - Algorithms:
    - To implement recursive algorithms
    - Graph search and traversal
  - Hardware and virtual machines
    - JVM is a *stack machine*

Applications of queues

- Queues are used in:
  - Operating systems:
    - To store messages exchanged by processes and threads
    - To store input/output events
    - For scheduling of processes
  - Simulation:
    - To manage simulation events
  - Data communication:
    - For buffering, for example buffers in network cards
  - Algorithms
    - Graph search and traversal

Stack Operations

- The main operations with stacks are:
  - STACK-PUSH(*S*,*x*) is the insertion operation for stacks.
  - STACK-POP(*S*) is the removal operation for stacks.
  - STACK-EMPTY(*S*) is an operation that checks if a stack is empty or not.
- Whenever we are trying to pop an element from an empty stack ⇒ *stack underflow*.
- Whenever we are trying to push an element onto a full stack ⇒ *stack overflow*.

Implementation of Stacks

- A stack can be implemented using an array *S*[1..*n*]. In the implementation there is a special field *top*[*S*] that is called the *top of the stack*. It represents the index of the element that has been inserted the most recently into the stack. Thus, at a certain moment in time the stack *S* contains the following elements *S*[1], *S*[2], …, *S*[*top*[*S*]].
- If *top*[*S*] = 0 then the stack is empty. Of course, initially we have to assure that *top*[*S*] is set to 0.
- Whenever we are trying to pop an element from a stack and *top*[*S*] = 0 ⇒ *stack underflow*.
- Whenever we are trying to push an element onto a stack and *top*[*S*] = *n* ⇒ *stack overflow*.
- Clearly, an underflow has to be always reported as an error. For an overflow we have two choices: either to report it as an error or to try to work out the situation by increasing dynamically the size of the array *S*. In what follows we ignore potential stack overflows.

Algorithms for stacks

```
STACK-EMPTY($S$)
1. return $top[S] = 0$
```

```
STACK-PUSH($S, x$)
1. $top[S] \leftarrow top[S] + 1$
2. $S[top[S]] \leftarrow x$
```

```
STACK-POP($S$)
1. if STACK-EMPTY($S$) then
2.     error "underflow"
3. else
4.         $top[S] \leftarrow top[S] - 1$
5.         return $S[top[S] + 1]$
```

Note that all the stack operations take a constant time.

Queue Operations

- The main operations with queues are:
  – QUEUE-PUSH(*Q*,*x*) is the insertion operation for queues.
  – QUEUE-POP(*Q*) is the removal operation for queues.
  – QUEUE-EMPTY(*Q*) is an operation that checks if a queue is empty or not.
- Whenever we are trying to pop an element from an empty queue $\Rightarrow$ *queue underflow*.
- Whenever we are trying to push an element onto a full queue $\Rightarrow$*queue overflow*.

Implementation of Queues

- A queue of maximum $n-1$ elements can be implemented using a circular array $Q[1..n]$ (i.e. the index following n is 1). In the implementation there are two special fields:
  - *front*[*Q*], index of element in front of the queue (the oldest element in the queue)
  - *rear*[*Q*], index of element at the rear of the queue where a new element will be inserted (one position before the newest element).
- The elements of the queue are (starting with the oldest one, ending with the newest one): *Q*[*front*[*Q*]], *Q*[*front*[*Q*]+1], …, *Q*[*rear*[*Q*]-1].
- The condition *front*[*Q*] = *rear*[*Q*] is used to signal that the queue is empty. Initially we have to set *front*[*Q*] and *rear*[*Q*] such that this condition holds.
- Whenever we are trying to pop an element from a queue with *front*[*Q*] = *rear*[*Q*] ⇒ *queue underflow.*
- Whenever we are trying to push an element onto a queue and *front*[*Q*] = *rear*[*Q*]+1 ⇒ *queue overflow.*

Algorithms for queues

Note that *length*[*Q*] is *n*. The comparisons with *n* are needed to implement the circular behavior of *Q*.

```
QUEUE-PUSH($Q, x$)
1. $Q[rear[Q]] \leftarrow x$
2. if $rear[Q] = length[Q]$ then
3.     $rear[Q] \leftarrow 1$
4. else
5.         $rear[Q] \leftarrow rear[Q] + 1$
```

```
QUEUE-POP($Q$)
1. $x \leftarrow Q[front[Q]]$
2. if $front[Q] = length[Q]$ then
3.     $front[Q] \leftarrow 1$
4. else
5.         $front[Q] \leftarrow front[Q] + 1$
6. return $x$
```

Note that all the queue operations take constant time.

Implementation of stacks (I)

The implementation contains 2 modules:

1. The *stack* module (files *stack.h* and *stack.c*).
2. The *main* module (file *main.c*).

We have added two more operations:

1. `stackInit()` to initialize the top of the stack.
2. `stackTop()` to return the value on top of the stack.

The program takes command lines from the standard input:

- `u` *number* to push the number onto the stack
- `o` to pop a number from the stack
- `p` to print the contents of the stack with the top shown to the left
- `x` to exit the program

Additionally, the program prints each command before executing it.

```
#ifndef STACK_H

#define STACK_H
#define STACK_SIZE 100

typedef struct stack {
  int top;
  int elements[STACK_SIZE];
} Stack;

void stackInit(Stack *s);
int stackEmpty(Stack *s);
void stackPush(Stack *s,
  int x);
int stackPop(Stack *s);
int stackTop(Stack *s);

#endif
```

Implementation of stacks (II)

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define error(x)
  fprintf(stderr,x);exit(1);

void stackInit(Stack *s) {
  s -> top = -1;
}

int stackEmpty(Stack *s) {
  return s->top == -1;
}

void stackPush(Stack *s,int x) {
  if (s->top<STACK_SIZE-1) {
    s->elements[++s->top] = x;
  }
  else {
    error("Stack overflow\n");
  }
}
```

```
int stackPop(Stack *s) {
  if (s->top>=0) {
    return s->elements[s->top--];
  }
  else {
    error("Stack underflow\n")
  }
}

int stackTop(Stack *s) {
  if (s->top>=0) {
    return s->elements[s->top];
  }
  else {
    error("Stack is empty\n")
  }
}
```

Implementation of stacks (III)

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

void stackPrint(Stack *s) {
  int i;

  i = s->top;
  while (i>=0) {
    printf("%3d",
      s->elements[i--]);
  }
  printf("\n");
}

main() {
  Stack s;
  int k;
  char cmdLine[80];
```

```
  stackInit(&s);
  gets(cmdLine);
  while (tolower(cmdLine[0])!='x') {
    puts(cmdLine);
    switch(tolower(cmdLine[0])) {
    case 'u':
      sscanf(cmdLine+1,"%d",&k);
      stackPush(&s,k);
      break;
    case 'o':
      k = stackPop(&s);
      break;
    case 'p':
      stackPrint(&s);
      break;
    }
    gets(cmdLine);
  }
}
```

A trace of the stack program

Input file:

```
u 12
u 45
p
o
u 14
p
u 7
p
o
o
p
u 9
p
u 5
u 10
p
o
o
o
o
p
```

Output file:

```
u 12
u 45
p
 45 12
o
u 14
p
 14 12
u 7
p
  7 14 12
o
o
p
 12
u 9
p
  9 12
u 5
u 10
p
 10  5  9 12
o
o
o
o
p

o
Stack underflow
```

Dynamic memory allocation (I)

- *Dynamic memory* = memory allocated at run-time.
- *Static memory* = memory allocated at compile-time.
- Programming languages like C and Pascal provide libraries for managing dynamic memory. This memory is allocated from a special area called *heap*.
- The heap is managed by a special language library component that implements functions like:
  - `new` and `dispose` in Pascal
  - `malloc()`, `calloc()` and `free()` in C.
- In languages that do not provide facilities for dynamic memory allocation it is possible to simulate it using static memory. Also, even if the pointer data type is not supported, it is possible to simulate it using integers.
- The idea is to define the heap as a static block of memory and define a set of operations for managing objects allocated in this block. To simplify things, we shall assume that all the objects have the same size (i.e. they are homogenous).

Dynamic memory allocation (II)

- We propose an implementation of dynamic memory allocation for fixed-size objects that uses the following variables:
  - The array *storage* for storing the objects;
  - The array *next* for storing links of the available objects;
  - An integer *available* that stores the index of the first available object. The available objects are linked together using the links stored in the array *next*.
- Initially all the objects in array *storage* are available. Whenever a request for allocating a new object is made, the object is taken from the *available* list. Whenever an object is released, it is pushed onto the *available* list. Thus, the available list behaves like a stack and object creation/deletion takes constant time.
- For representing NIL we are using a value that is not a valid index in *storage* (i.e. if *storage* has indices starting with 0 we can take NIL = -1, or if it has indices starting with with 1 we cam take NIL = 0).
- If *available* = NIL then the allocation of a new object fails. This means that all the objects of the *storage* array have already been allocated before.

Algorithms for dynamic memory allocation

```
OBJECT-INIT()
1. for $i = 1$ to $size[storage] - 1$ do
2.     $next[i] \leftarrow i + 1$
3. $next[size[storage]] \leftarrow$ NIL
4. $available \leftarrow 1$
```

```
OBJECT-NEW()
1. if $available =$ NIL then
2.     return NIL
3. else
4.     $x \leftarrow available$
5.     $available \leftarrow next[x]$
6.     return $x$
```

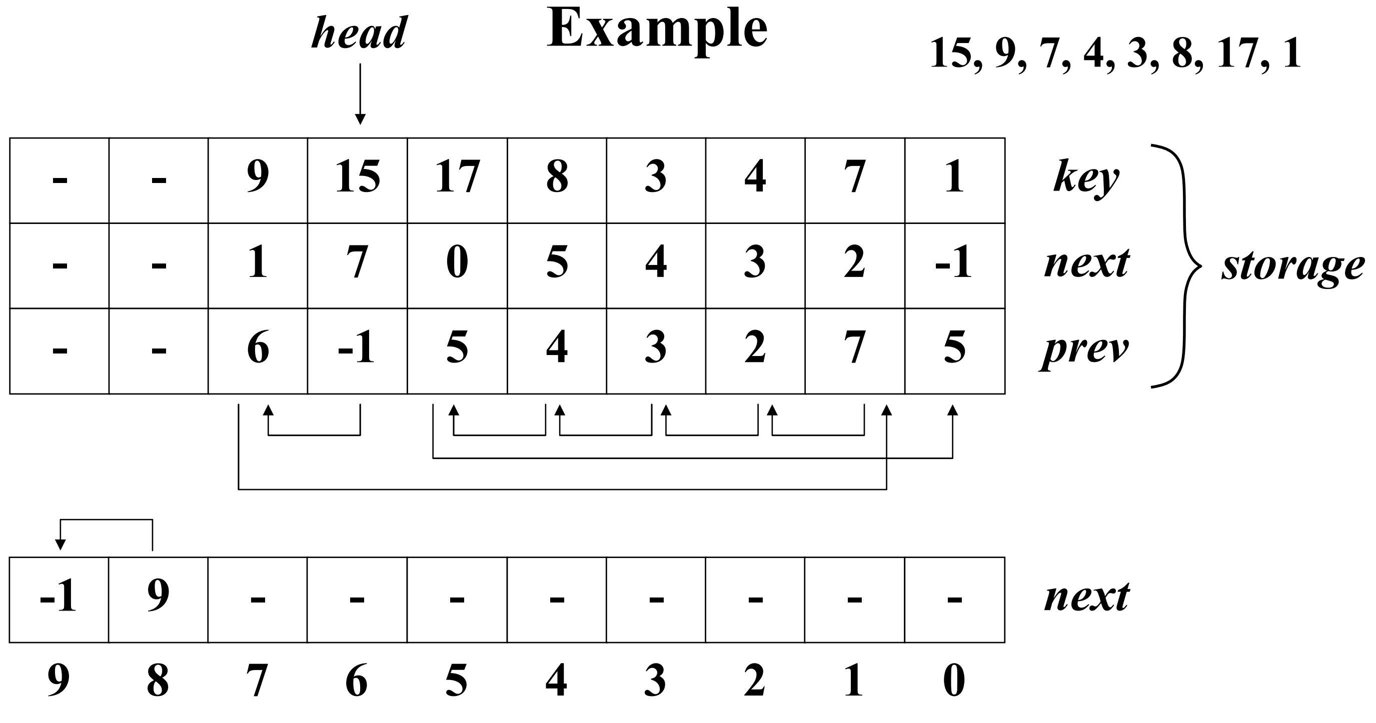
We assume that the arrays *next* and *storage* have indices from 1 to *n* and that *n* = *size*[*storage*]. NIL = -1

2. OBJECT-NEW and OBJECT-DELETE can be implemented such that they use pointers to objects instead of integer indices.

```
OBJECT-DELETE($x$)
1. $next[x] \leftarrow available$
2. $available \leftarrow x$
```

Example

15, 9, 7, 4, 3, 8, 17, 1



-	-	9	15	17	8	3	4	7	1	*key*
-	-	1	7	0	5	4	3	2	-1	*next*
-	-	6	-1	5	4	3	2	7	5	*prev*

-1	9	-	-	-	-	-	-	-	-	*next*
9	8	7	6	5	4	3	2	1	0	

In this example the objects are nodes of a doubly-linked list. The links shown in the array *storage* follow the *next* field of a node of this list.

In this example array subscripts start with 0 !

Implementation of dynamic memory allocation (I)

- We consider the implementation of dynamic memory allocation for the nodes of a doubly-linked list.
- Requirement: the implementation must be done such that the implementation of doubly-linked lists shown in the previous lecture will need only minimal changes:
  - The *list* module (files *list.c* and *list.h*) will not be changed at all;
  - The main program will be changed only where the calls to dynamic memory allocation functions were made.
- The allocator is implemented as a separated module *oalloc* (files *oalloc.c* and *oalloc.h*). This module must know the type of the objects (`ListNode`) so it will include the header *list.h*.
- The main program must initialize and then call the allocator. So it will include the header *oalloc.h*.
- Conclusion: in this way the interaction between the original program and the allocator is minimized.
- Note that because we don't want to change the *list* module the allocator must return pointers, not integer indices.

Implementation of dynamic memory allocation (II)

```
#include <stdlib.h>
#include "oalloc.h"
void storageInit(void) {
  int i;
  for (i=0;i<N_OBJECTS-1;i++) {
    next[i] = i+1;
  }
  next[N_OBJECTS-1] = NIL;
  available = 0;
}
ListNode* objectNew(void) {
  if (available == NIL) {
    return NULL;
  }
  else {
    int x = available;
    available = next[x];
    return &storage[x];
  }
}
void objectDelete(ListNode *x) {
  int y = x-storage;
  next[y] = available;
  available = y;
}
```

```
#ifndef OALLOC_H

#define OALLOC_H

#include "list.h"

#define N_OBJECTS 100
#define NIL -1

ListNode storage[N_OBJECTS];
int next[N_OBJECTS];
int available;

void storageInit(void);
ListNode* objectNew(void);
void objectDelete(ListNode *);

#endif
```

Implementation of dynamic memory allocation (III)

```c
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
#include "oalloc.h"

void printList(List l) {
  ListNode *n;
  n = l.head;
  while (n != NULL) {
    printf("%3d",n->key);
    n = n->next;
  }
  printf("\n");
}

main() {
  List l;
  int k;
  char cmdLine[80];
  ListNode *n = NULL;
  storageInit();
  l.head = NULL;
  gets(cmdLine);
  while (tolower(cmdLine[0]) != 'x') {
    switch(tolower(cmdLine[0])) {
    case 'i':
      sscanf(cmdLine+1,"%d",&k);
      n = objectNew();
      n->key = k;
      listInsert(&l,n); break;
    case 'r':
      sscanf(cmdLine+1,"%d",&k);
      n = listSearch(&l,k);
      if (n != NULL) {
        listRemove(&l,n);
        objectDelete(n);
      }
      break;
    case 'p':
      printList(l); break;
    }
    gets(cmdLine);
  }
}
```

The lines that have been underlined show the points where the original main program has been changed.

Homework

1. We consider an arithmetic expression with operands, operators * and + and parentheses '(' and ')'. Operands can be:
   – variables represented by single letters and
   – numbers represented by single digits 0, …, 9.

   The program yakes as input:
   – A string representing a syntactically correct expression
   – The values of variables. A value can be only a digit.

   Design an algorithm and develop a C program that reads an expression together with the values of the variables and determines the value of the expression.