Abstract data types

October 16, 2019 8:02 AM

Circular linked lists

Stack ADT

- mathematical description of an object and a collection of data and the operations for accessing the data
- ex. Dictionary ADT
  - stores pairs of srings
  - operations:
    - Insert
    - Remove
    - Lookup
- Implementing ADTs
  - ex. RPN Notation, postfix notation
  - A mathematical notaion where every operator follows its operands
  - ex. Infix: 5+(1+2)*4)-3, RPN: 5 1 2 + 4 * + 3 -
  - read string left to right, everytim see number left to right, store insie containre
  - Apply + to last 2 operand, Apply * to last 2 opernands
  - postfix string contains integers and characters but data collection contains only ints
  - if is operand, store (operand)
  - if symbol is operater
    - RHS=remove();
    - LHS = remove();
    - result LHS operator RHS;
    - store(result)
  - result = remove()
- Describing ADT
  - items never inserted between existing items
  - LIFO (Last in, first out)
  - top of stack at end, other end is bottom
- Operations:
  - **push:** insert at top
  - **pop:** remove and return top item
  - **peek:** return top item
  - **isEmpty:** does the stack contain any item
- order of items based on order which they arrive
- ex. use stack to explore every branch in maze
  explore branch, use stack to backtrack if deadend

Stack implementation

- can use arrays, linked lists, etc.
- treat last element of array as "top" of the stack
- information to track:
  - index of top of item (first free space)
  - maximum size of array
- 3 functions:
  - initialize
  - check if empty
  - check full
- "top" = index top element
- main

  int main() {

  Stack mystack;

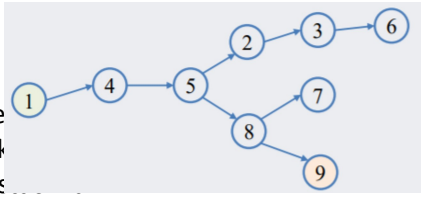
  initialize(&mystack) // points to empty array in heap

```c
int square(int x) {
  return x*x;
}

int squareOfSum(int x, int y) {
  return square(x+y);
}

int main() {
  int a = 4;
  int b = 8;
  int total = squareOfSum(a, b);
  printf("Total: %d\n", total);
  return 0;
}
```

```
    push&mystack);
    pop(&mystack);
}
```



- pushing onto stack: tew top, increment top before
- pop: top item removed, stack shrinks, ensure stack
- array created with inital size, push returns false if s
- if need to push additional items, reallocate larger array

```
typedef struct {
  int top; // index of first free space
  int capacity; // maximum size of array
  int* arr; // pointer to array (in dynamic memory)
} Stack;
```

Quiz

- Stacks, use push/pop
- grab whatever is on top

Stack implementation

- array created with initial size, push returns false if
- reallocate larger array

```
void initialize(Stack* st) {
  st->top = -1;
  st->capacity = 8; // or some other value
  st->arr = (int*) malloc(capacity * sizeof(int));
}
```

stack

```
int push(Stack* st, int val) {
  int i; // used for reallocation
  int* newarr;
  if (st->top == st->capacity - 1) {
    // reallocate a larger array
    st->capacity = 2 * st->capacity;
    newarr = (int*) malloc(st->capacity * sizeof(in
    for (i = 0; i <= st->top; i++)
      newarr[i] = st->arr[i];
    free(st->arr);
    st->arr = newarr;
  }
  // continue with push
  st->top++;
  st->arr[st->top] = val;
  return TRUE;
}
```

```
#def
#def
```

```
int isEmpty(Stack* st) {
  if (st->top == -1)
    return TRUE;
  else
    return FALSE;
}
```

```
int isFull(Stack* st) {
  if (st->top == st->capacity - 1)
    return TRUE;
  else
    return FALSE;
}
```

```
typedef struct {
  int top;
  int capacity;
  int* arr;
} Stack;
```

```
int push(Stack* st, int value) {
  if (!isFull(st))
    st->top++;
    st->arr[st->top] = value;
    return TRUE;
  else
    return FALSE;
}
```

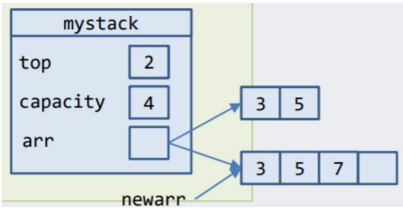
- know it's full when top = capacity
- use a loop to copy elements into new array
- struct arr pointer broken with array1, points to larger new array

```
int peek(Stack* st) {
  if (!isEmpty(st))
    return st->arr[st->top];
  else
    return FALSE;
}
```

```
int pop(Stack* st) {
  if (!isEmpty(st))
    st->arr[st->top] = -1;
    st->top--;
    return TRUE;
  else
    return FALSE;
}
```

Complexity of array resizing

- stack with n capacity completly full
  - complexity of 1 push operation is O(n), else O(1) for regular push
- stack capacity of n completely empty
  - complexity of 2n push operation is O(n)
  - +n more pushes at O(1)
  - Therefore, 2n insertion at O(1), 1 reseize at O(n) = 3n, (On)
- avg complexity of single push operation?
  - 1 insertion O(1)
  - copy 1 + 1 more insertion at O(1)
  - copy 2+2 more insertions at O(!)
  - therefore O(1)
- total number of elements copied = n = O(n) for array to be full
  - 2n cost total/ ni instertions
- ex.
  - O: one insertion at O(1)
  - XO: copy 1, one insertion at O(1)
  - XXO: copy 2, one insertion at O(1)
  - XXXO: copy 3, one insertion at O(1)
  - for copy n-1 + 1 more insertion, total n insertion at O(1) each
    - total n insertions at O(1) each: 1+2+#+...+n-1 elements copied
    - sum of $n(n-1)/2 = O(n^2)$
  - total cost for n insertions = $O(n) + O(n^2)$



- average cost per insertion = $O(n^2)/n$ = O(n) per insertion

Stacks and linked lists

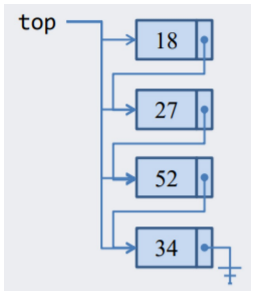
- usage for stack involves only calling stack functions (push, pop, peek)
- data storage can be implemented with other data structure
- with a singly linked list, front of list is accessed easily
  - stack inserts and removes from the top, insert and remove from the front of the list
  - point top to node bellow and call free after

typedef struct Stacked {
linkedList ll;
} Stacked;

int Push (stacked, st, int, val) {*
InsertFront(&(st->ll), val);
}


```
struct Node {
  int data;
  struct Node* next;
};
```

```
typedef struct {
  struct Node* top;
} Stack;
```




```
void initStack(Stack* st) {
  st->top = NULL;
}
```

```
int isEmpty(Stack* st) {
  if (st->top == NULL)
    return TRUE;
  else
    return FALSE;
}
```