CS 2412 Data Structures

Chapter 2

Stacks and recursion

2.1 Stacks

A stack is a data structure in which all insertions and deletions of entries are made at one end, called top of the stack.

Examples:

- A stack of files.
- DOS command line

LIFO: last in first out (or first in last out FILO).

Stacks are very important data structure. In any kind of computer, there is a stack area in memory (usually in higher memory addresses).

When a subprogram is called, a return address is put in the stack. In this way, the computer can return to the place after the subprogram is finished.

Main operations of stack

- Creation (create a stack).
- Clearance (damage a stack).
- Push (add an element to the stack)
- Pop (get an element from the stack)

Specification for a stack

To specify a stack, we consider the required functions and the preconditions and post-conditions for each function.

We will use type `Stack` and `StackEntry` to denote the type stack and the entries of the stack.

In general, for a stack, all the entries should be the same type data.

Initialization:

`Void CreateStack(Stack *s);`

precondition: None.

postcondition: The stack `s` has been created and is initialized to be empty.

Status:

`Boolean StackEmpty(Stack *s);`

precondition: The stack exists and it has been initialized.

postcondition: Return `TRUE` if the stack is empty, `FALSE` otherwise.

`Boolean StackFull(Stack *s);`

precondition: The stack exists and it has been initialized.

postcondition: Return `TRUE` if the stack is full, `FALSE` otherwise.

Basic operations;

`void Push(StackEntry item, Stack *s);`

precondition: The stack exists and it is not full.

postcondition: The argument `item` has been stored at the top of the stack.

`void Pop(StackEntry *item, Stack *s);`

precondition: The stack exists and it is not empty.

postcondition: The top of the stack has been removed and returned in `*item`.

Other operations:

```
int StackSize(Stack *s);
```

precondition: The stack exists and it has been initialized.

postcondition: The function returns the number of entries in the stack.

```
int StackTop(StackEntry *item, Stack *s);
```

precondition: The stack exists and it is not empty.

postcondition: The item at the top of the stack is returned without being removed.

(Return an integer to indicate the state of precondition).

Termination

```
void ClearStack(Stack *s);
```

precondition: The stack exists and it has been initialized.

postcondition: All entries in the stack have been deleted; the stack is empty.

Note: The postcondition can be defined to: deleted all the entries and the stack itself.

Traversable stack: This is not a ordinary stack and the following operation is not an operation of a stack.

```
void TraverseStack(Stack *s, void(*Visit)());
```

precondition: The stack exists and it has been initialized.

postcondition: The function that `Visit` points to, has been invoked for each entry in the stack, beginning with the entry at the top and proceeding toward the bottom of stack.

Implementation of stacks

Declarations

```
#define MAXSTACK 10 //size of the stack
typedef char StackEntry; //stack of char
typedef struct {
  int top;
  StackEntry entry[MAXSTACK];
} Stack;
```

```
void Push(StackEntry item,Stack *s)
{
 if(StackFull(s))
   Error("Stack is full");
 else
   s->entry[s->top++]=item;
}


void Pop(StackEntry *item, Stack *s)
{
 if(StackEmpty(s))
    Error("Stack is empty");
 else
    *item=s->entry[--s->top];
}
```

```
Boolean StackEmpty(Stack *s)
{
  return s->top <=0;
}

Boolean StackFull(Stack *s)
{
  return s->top >=MAXSTACK;
}

void CreateStack(Stack *s)
{
  s->top=0;
}
```

Dynamically allocating memories:

```
void *calloc(n,size)
void free(p)
void *malloc(size)
void *realloc(p,size)
```

Example:

```
tp = (double*) malloc(n*sizeof(double));
if (tp==NULL)
printf("could not allocate %d doubles\n",n);
```

- Don't assume `malloc` will always succed.
- Don't assume the storage `malloc` provides is initialized to zero.
- Don't modify the pointer returned by `malloc`.
- `free` only pointers obtained from `malloc`, and don't access the storage after it's been freed.

Linked stack

Dynamic memory allocation can allocate memories during the run time and free the memories after use.

```
typedef struct node {
  void* dataPtr;
  struct node* link;
 } STACK_NODE;

typedef struct {
  int count;
  STACK_NODE* top;
 } STACK;
```

```
STACK* createStack (void) {
  STACK* stack;
  stack = (STACK*)malloc(sizeof (STACK));
   if (stack) {
     stack->count = 0;
     stack->top = NULL;
    }
  return stack;
}
```

```
bool pushStack (STACK* stack, void* dataInPtr) {
 STACK_NODE* newPtr;
 newPtr = (STACK_NODE*)malloc(sizeof(STACK_NODE));
 if (!newPtr)
   return false;
 newPtr->dataPtr = dataInPtr;
 newPtr->link = stack->top;
 stack->top = newPtr;
 (stack->count)++;
 return true;
}
```

```
void* popStack (STACK* stack) {
  void* dataOutPtr;
  STACK_NODE* temp;
  if(stack->count == 0)
   dataOutPtr = NULL;
  else {
    temp = stack->top;
    dataOutPtr = stack->top->dataPtr;
    stack->top = stack->top->link;
    free (temp);
    (stack->count)--;
    }
  return dataOutPtr;
}
```

```
void* stackTop (STACK* stack) {
  if (stack->count == 0)
    return NULL;
  else
    return stack->top->dataPtr;
}

bool emptyStack (STACK* stack) {
  return (stack->count == 0);
}
```

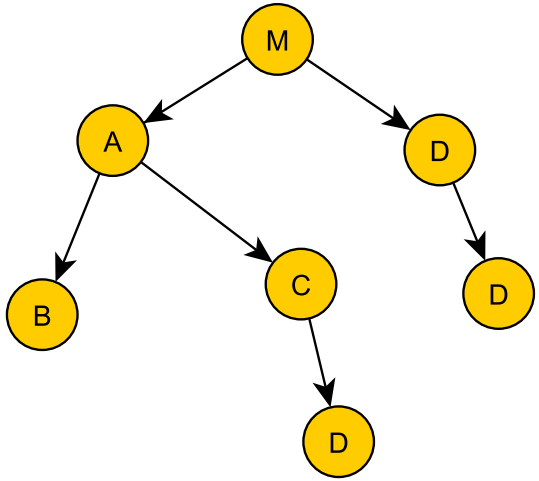
```
bool fullStack(STACK* stack) {
  STACK_NODE* temp;
    if((temp=(STACK_NODE*)malloc(sizeof (*(stack->top)))))
     {
      free (temp);
      return false;
      }
 return true;
}

int stackCount (STACK* stack) {
 return stack->count;
}
```

```
STACK* destroyStack(STACK* stack) {
 STACK_NODE* temp;
 if (stack) {
   while(stack->top !=NULL) {
     free(stack->top->dataPtr);
     temp = stack->top;
     stack->top = stack->top->link;
     free (temp);
     }
   free (stack);
 return NULL;
}
```

2.2 Introduction to recursion

Stack frames for subprograms

Suppose we has 4 functions A, B, C and D. The main function M first calls A and A needs to call B. Then A calls C and C calls D. Finally, M calls D, D calls D.



Computer uses stack frames in memory to remember the return places.

Recursion

Recursion is the name for the case when a subprogram invokes itself or invokes a series of other subprograms that eventually invokes the first subprogram again.

Stack and tree of subprogram calling

Theorem During the traversal of any tree, vertices are added to or deleted from the path back to the root in the fashion of a stack. Given any stack, conversely, a tree can be drawn to portray the life history of the stack, as items are pushed onto or popped from it.

Example:

To implement factorial function, there are two different ways to do it.

```
int Factorial(int n)
{
 int i,k=1;
 if(n==0)
   return 1;
 else
   for(i=1;i<=n;i++)
    k=k*i;
}
```

Using recursion:

```
int Factorial(int n)
{
  if(n==0)
    return 1;
  else
    return n*Factorial(n-1);
}
```

The source code looks more simple if a recursion is used. But recursion uses more memory.

The towers of Hanoi

There are 3 diamond needles. On the first needle are stacked 64 golden disks, each one slightly smaller than the one under it. The task is to move all the golden disks from the first needle to the third, subject to the conditions that only one disk can be moved at a time, and that no disk is ever allowed to be placed on top of a smaller disk.

We describe the problem as

$$\text{Move}(64;1,3,2)$$

It means move 64 disk from 1 to 3 using 2 as auxiliary needle.

Use recursion solution

Question: In what condition we can move the last disk to the third needle?

Answer: It must be the case that the third needle is empty, the first needle only has the last disk and all others are placed on the second needle.

That means we must have done

$$\text{Move}(63;1,2,3)$$

After that we just need to do the following things:

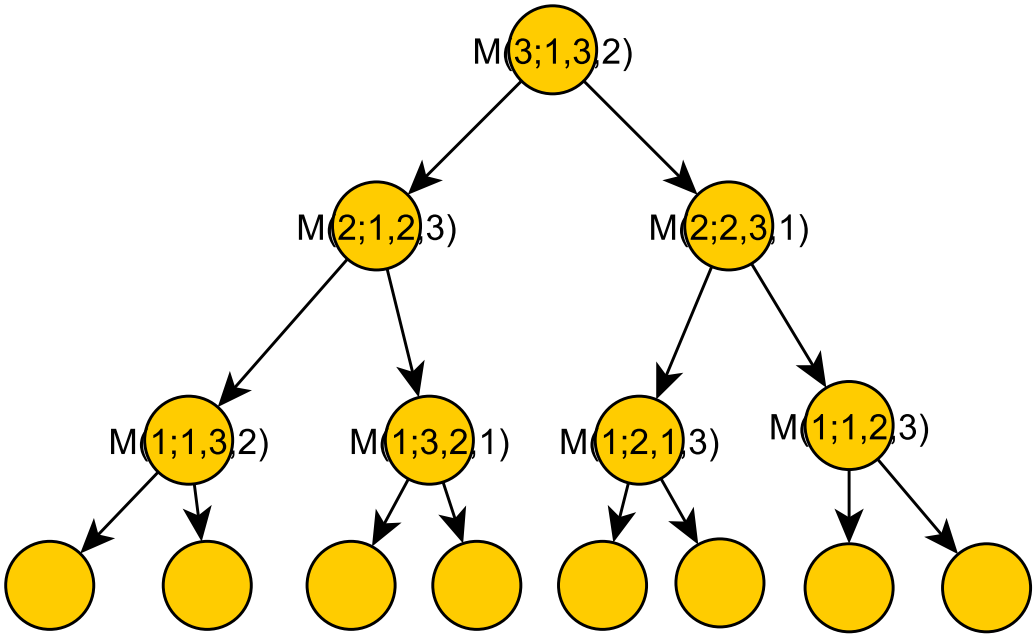
- Move the last disks to the third needle.
- Move other disks from second needle to the third needle, i.e.,

$$\text{Move}(63;2,3,1)$$

```
#define DISK 64

int main(void)
{
 Move(DISK,1,3,2);
 return 0;
}

void Move(int count, int start, int finish, int temp)
{
  if(count > 0)
  Move(count-1,start,temp,finish);
  printf("Move a disk from %d to %d.\n",start,finish);
  Move(count-1,temp,finish,start);
}
```

To study a recursion function, try a very small example to trace its action is usually useful.

We can use tree to see the cases of DISK equals to 2 and 3. The tree for 3 disk is as follows.



When DISK == 64, the number of non-leaves is

$$1 + 2 + 4 + \cdots + 2^{63} = 2^0 + 2^1 + 2^2 + \cdots + 2^{63} = 2^{64} - 1$$

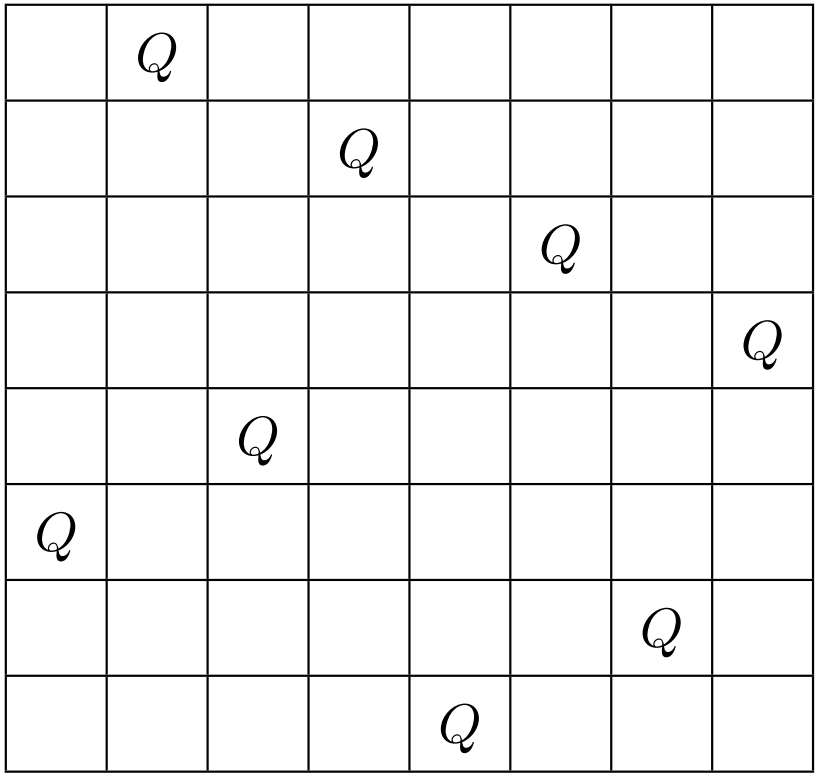
That means there are $2^{64} - 1$ moves to finish the towers of Hanoi problem.

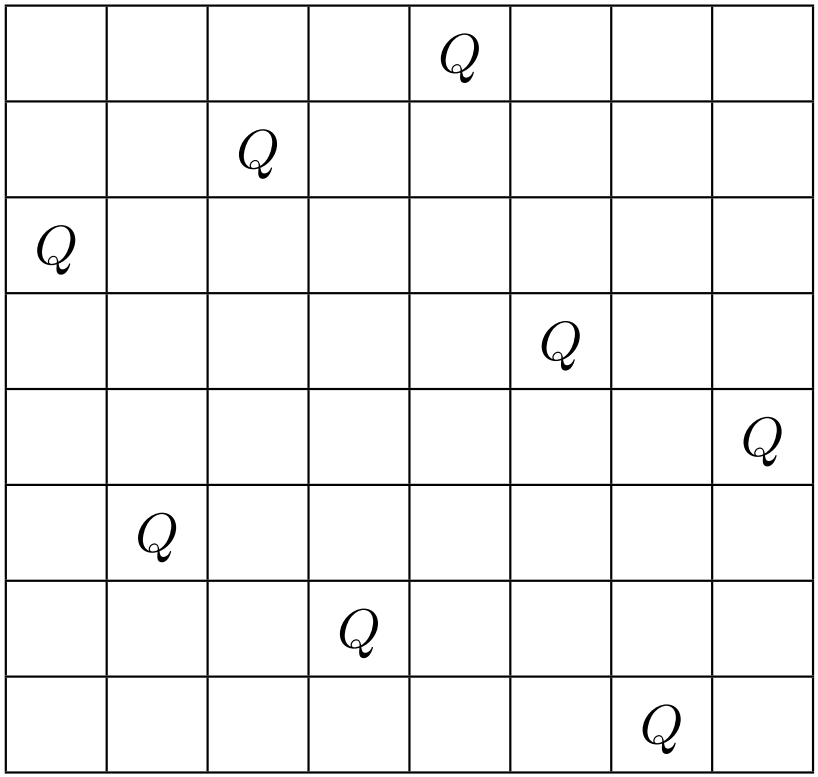
It is easy to estimate that $2^{64} > 1.6 \times 10^{19}$. If we can move one disk at one second, then it will take about 5×10^{11} years. I.e., it will take about 50 billion years!

On the other hand, the space for keeping track of 64 recursive calls is not big.

2.3 Backtracking

A puzzle: How to place eight queens on a chessboard so that no one can take another.





How to find one solution? How to find all the solutions?

Example of four queens:

Q	x	x	x
x	x		
x		x	
x			x

x	Q	x	x
x	x	x	Q
Q	x		x
	x	Q	x

Backtracking algorithms Used in a search problem by constructing partial solutions and then trying to extend the partial solutions. The main steps:

- Establish a partial solution of the search problem. Make sure that the partial solutions remain consistent with the requirement of the problem.
- Extend the partial solutions toward the completion. If a inconsistency occurs, then the algorithm backs up by removing the most recently constructed part of the solution and try another possibility.

Algorithm for eight-queen puzzle

Try to use recursion. Define a function `AddQueen` which add a queen in an unguarded position on the board. We can recursively call this function until a solution is found, or backtrack a position.

We use `queencount` to denote the number of queens have put on the board. The pseudo code of the function can be:

```
void AddQueen(void)
{
 for(each unguarded position p on the board) {
   place a queen in position p;
   queencount++ and mark guarded positions;
   if (queencount==8)
      print the configuration;
   else
     AddQueen();
   end if
   remove the queen from position p;
   queencount-- and free guarded positions;
   }
}
```

To implement the algorithm, we need more specification.

`int col[8]` is used to record the position of queens. `col[i] =` the column the ith queen is in.

`Boolean colfree[8]` is used to record if the column is unguarded.

For diagonals, it is more difficult to record and check. We can use a 8×8 array to record the guarded positions, but we have a better method.

It is noted that for the position (i, j) all the positions in the downward diagonal the difference of row and column indices is the same $(i - j)$. Similarly the sum of the row and column indices is the same for all the positions in the upward diagonal. So `Boolean downfree[15]` is used to record the unguarded positions on the downward diagonals and `Boolean upfree[15])` is used to record the unguarded positions on the upward diagonals.

2.4 Principles of recursion

Design a recursion algorithm:

- Find the key step: Find a step which will be followed with same or similar steps towards the solution.
- Find a stopping rule: Each recursion algorithm has to have a stopping rule.
- Outline the algorithm: Abstraction, specification, details.
- Check termination: To avoid infinite loops.
- Draw a recursion tree: To analysis the algorithm.

From a recursion tree, we can estimate the requirements for space and time for running the program.

The depth of the tree indicated the space requirement. The number of nodes related to the time requirement.

Tail recursion

If the last-executed statement of a function is recursive call to the function itself, then this call can be eliminated by reassigning parameters to the values specific in the recursive call, and then repeating the whole function.

The algorithm of Towers of Hanoi:

```
void Move(int count, int start, int finish, int temp)
{
 int swap;
 while(count > 0) {
   Move(count-1,start,temp,finish);
   printf("Move disk %d from %d to %d.\n",count,start,
     finish);
   count--;
   swap=start;
   start=temp;
   temp=swap;
 }
}
```

Recursion and iteration

Fibonacci number:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, \text{ for } n \geq 2.$$

```
int Fibonacci(int n)
{
 if (n<=0)
   return 0;
 else if (n==1)
   return 1;
 else
   return Fibonacci(n-1) + Fibonacci(n-2);
}
```

```
int Fibonacci(int n)
{
 int i;
 int twoback,oneback,current;
 if (n<=0) return 0;
 else if (n==1) return 1;
 else {
   twoback = 0; oneback = 1;
   for(i=2;i<=n;i++){
     current=twoback+oneback; twoback=oneback;
     oneback=current;
    }
   return current;
  }
}
```

Make decision of using recursion by recursion tree.

- If the recursion tree has a simple form, the iterative version may be better.
- If the recursion tree involves duplicate tasks, then data structures other than stacks will be appropriate, and the need for recursion may disappear.
- If the recursion tree appears quite bushy, with little duplication of tasks, then recursion is likely the nature method.