Stacks and Queues

Biostatistics 615
Lecture 11
Last Lecture ...

- Unraveled Quick Sort
  - Divide-And-Conquer algorithm
    - Typically recursive
  - Used explicit stack to track computations
    - Replaced recursion with loop
  - Provided guarantee on maximum memory use
Quick Sort

Array is successively subdivided, around partitioning element.

Within each section, items are arranged randomly
Last Lecture: Merge Sort

- Divide-And-Conquer algorithm
  - \( N \log N \) cost in computing time
  - Auxiliary array with \( N \) elements also required

- Can be unraveled without explicit stack
  - Build small sorted arrays of size \( 2^k \) which are merged into arrays of size \( 2^{k+1} \)
Merging Pattern for $N = 21$

This pattern results from the recursive formulation.
Merging Pattern for \( N = 21 \)

This pattern results from the iterative formulation.
Merge Sort in Action
Today

- Abstraction
  - Hide implementation from users
  - Structures (in C)
  - Lists (in R)

- Examine alternative implementations for a stack
Abstraction in Computers

- Physical properties of silicon chips
- Sequence of 0 – 1 bits
- "Machine" that manipulates bits
- Programming language
- Program that tackles a specific task
- Complex program made of many parts
Abstraction in Programming

- Hide specifics of object from users
- Define and document an interface
  - The exact details of the implementation can change, as better algorithms are developed
- The basis for reusable code
Union-Find Algorithms

- Recall algorithms we applied to connectivity problem

- Class algorithms based on union-find operations
  - "FIND" operation checks connectivity
  - "UNION" operation incorporates information on a new connection
Checking Connectivity

RandomConnections <- function(N, M)
{
    UFdata <- UFinit(N);

    for (i in 1:M)
    {
        p = sample(1:N, 1)
        q = sample(1:N, 1)

        if (UFind(UFdata, p) != UFind(UFdata, q))
        {
            UFdata <- UFUnite(UFdata, p, q);
            cat("New connection:", p, q, "\n");
        }
    }
}
Checking Connectivity

```c
void RandomConnections(int N, int M)
{
    int p, q; UF_info * UF_data;

    UF_data = UF_init(N);

    for (i = 0; i < M; i++) {
        p = rand() % N;
        q = rand() % N;

        if (UF_find(UF_data, p) != UF_find(UF_data, q)) {
            UF_unite(UF_data, p, q);
            printf("New connection: %d to %d\n", p, q);
        }
    }
}
```
Abstraction

- Specific data structures are hidden in `UF_data`

- This variable could group multiple pieces of information
  - As a list in R
  - As a structure (or pointer) in C
  - As a class in C++
R lists

- Created with `list()` function
  - Alternatively, with `vector(mode = "list", ...)`

- Collection of objects
  - Each can have a different type
  - Objects can be lists, arrays, or simple values

- Accessed with `[[i]]` index or `$` operator
R Lists: Example

```r
fred <- list(
  name = "Fred",
  wife = "Mary",
  children = 3,
  ages = c(4,7,9))

fred$wife
fred[[2]]
```
UF Implementation in R

UFinit <- function(N)
{
  return(list(id = 1:N, wt = rep(1, N)))
}

UFfind <- function(data, i)
{
  while (i != data$id[i]) i <- data$id[i];
  return(i)
}
UF Implemente in R

UFunite <- function(data, p, q)
{
  p = UFfind(data, p);
  q = UFfind(data, q);

  if (data$wt[p] > data$wt[q]) {
    data$id[q] <- data$id[p];
    data$wt[p] <- data$wt[p] + data$wt[q]
  } else {
    data$id[p] <- data$id[q];
    data$wt[q] <- data$wt[p] + data$wt[q]
  }

  return(data)
}
C Structures

- **Use the `struct` keyword**
  - Analogous to lists in R

- **Since C is a strongly typed language**
  - Must be declared
  - Each item must have a type

- **Contents accessed with "." or "->" operators**
C Structures

typedef struct {
    int x;
    int y;
} point;

point origin;
origin.x = origin.y = 0;

point * ptr;
ptr->x = 0; ptr->y = 1;
typedef struct {
    /* Used to track grouping for each item */
    int * id;

    /* Used to track weights for each node */
    int * wt;
} UF_info;
UF Implementation in C

UF_info * UF_init(int N)
{ int i; UF_info * data = malloc(sizeof(UF_info));

    data->wt = malloc(sizeof(int) * N);
    data->id = malloc(sizeof(int) * N);

    for (int i = 0; i < N; i++)
        { data->wt[i] = 1; data->id[i] = i; } 

    return data;
}

int UF_find(UF_info * data, int i)
{ 
    while (i != data->id[i]) i = data->id[i];
    return i;
}
void UF_unite(UF_info * data, int p, int q)
{
    p = UF_find(data, p);
    q = UF_find(data, q);

    if (data->wt[p] > data->wt[q])
    {
        data->id[q] = data->id[p];
        data->wt[p] += data->wt[q];
    } else {
        data->id[p] = data->id[q];
        data->wt[q] += data->wt[p];
    }
}
So far ...

- Use lists (in R) or structures (in C) to group related data

- Avoids:
  - Long argument lists
  - Global variables

- Hides implementation details from caller
Collections of Objects

- A simple abstraction would allow:
  - Initializing the collection
  - Checking whether the collection is empty
  - Adding an element to collection
  - Removing an element from collection

- Stacks and Queues
Pushdown Stack

- Elements are handled in LIFO fashion:
  - Last In, First Out
  - Most recently added element returned

- An example:
  - Function calls are managed through a pushdown stack
Operations on a Stack

- stackInit()
  - Initializes data structures
- stackPush()
  - Adds item to collection
- stackPop()
  - Removes most recently added item
- stackPeek()
  - Retrieves item at top of stack
- stackEmpty()
  - Checks if stack is empty
Postfix Calculator

- A Simple Stack Application

- Sequence of numbers and operators
  - Numbers pushed onto stack
  - Operators (*, +, etc.) result in the sequence
    - Pop two values
    - Evaluate expression
    - Push result onto stack
Stack Based Postfix Calculator

Stack * s = stackInit(N);
for (i = 0; i < N; i++)
{
    if (a[i] == '+')
        stackPush(s, stackPop(s) + stackPop(s));
    else if (a[i] == '*')
        stackPush(s, stackPop(s) * stackPop(s));
    else if (a[i] >= '0' && a[i] <= '9')
    {
        stackPush(s, 0);
        while (a[i] >= '0' && a[i] <= '9')
            stackPush(s, 10*stackPop(s) + a[i++] - '0');
        i--;
    }
}
Postfix Calculator

- Just one example of a program that could use stack functionality
- Quicksort is another example we have seen
- Many others
A Simple Stack Implementation

- Could be based on arrays
- Items are added to the end
- Items are also removed from the end
The Stack as An Array

- Simple implementation
- Uses an array to represent stack
  - Maximum stack size must be pre-specified

- Track location of top item in the stack
  - Why do we add to the end?
  - Would adding to the beginning also work?
Array-based Stack Structure

typedef struct {
    /* Used to track items in collection */
    Item * s;

    /* Tracks the total number of items */
    int N;
}

Stack;
An Array Based Stack...

Stack * stackInit(int maxItems)
{
    Stack * data = malloc(sizeof(Stack));
    data->s = malloc(sizeof(Item) * maxItems);
    data->N = 0;
    return data;
}

void stackPush(Stack * stack, Item item)
{ stack->s[stack->N++] = item; }

Item stackPop(Stack * stack)
{ return stack->s[--(stack->N)]; }

int stackEmpty(Stack * stack)
{ return stack->N == 0; }
Stack as a Chain of Pointers

- Each element points to the next
  - Fixed overhead per element

- Memory is allocated "as needed"

- Maximum size does not have to be known in advance
Pushing onto The Stack

Stack with element 10

Stack after pushing 11
Popping from The Stack

Stack after popping one element

start

1

10
/* Declares type, but defers details for later */
struct node;

/* Declares type for pointing to nodes */
typedef struct node * nodePtr;

/* Each node contains item and pointer to the next */
struct node {
    Item   item;
    nodePtr next;
};

/* Stack points to first node */
typedef struct {
    nodePtr start;
} Stack;
Creating a New Node ...

```c
nodePtr newNode(Item item, nodePtr next) {
    nodePtr x = malloc(sizeof(struct node));
    x -> item = item;
    x -> next = next;
    return x;
}
```
A Pointer Based Stack...

```c
Stack * stackInit(int maxItems)
{ Stack * data = malloc(sizeof(Stack));
  data->start = NULL;
  return data; }

void stackPush(Stack * stack, Item item)
{ stack->start = newNode(item, stack->start); }

Item stackPop(Stack * stack)
{ Item item = stack->start->item;
  nodePtr s = stack->start->next;
  free(stack->start); stack->start = s;
  return item; }

int stackEmpty(Stack * stack)
{ return stack->start == NULL; }
```
Implementation in R

- Dynamic arrays provide a good starting point
- Implementing Pop() is tricky:
  - Requires changing array and returning a value
  - R functions can't conveniently change multiple variables
  - Retrieve element with call to Peek() first.
An Implementation in R

```r
stackInit <- function(maxSize = INF)
  { return(c()); }
stackPush <- function(stack, x)
  { return(c(x, stack)) }
stackAlmostPop <- function(stack)
  { return(stack[-1]) }
stackPeek <- function(stack)
  { return(stack[1]) }
stackEmpty <- function(stack)
  { return(length(stack)) }
```
Other General Collections

- FIFO Queues
  - First In, First Out

- Linked Lists
  - Similar to stack managed through pointers
  - Collection of elements linked by pointers to next (and optionally) previous element
Today

- Abstraction
  - Make different portions of code as independent as possible

- Structures and Lists

- Alternative representations for stacks
Further Reading

- Sedgewick, Chapter 4