LINKED LISTS, STACKS AND QUEUES

Lecture notes of the course "*Programming Techniques*"

Lê Hồng Phương[1]

[1]Department of Mathematics, Mechanics and Informatics
Hanoi University of Science, VNUH
<*phuonglh@gmail.com*>

10/2012

Content

Content

Content

Content

Content

Content

Structures

- A structure is a collection of related variables (of possible different types) grouped together under a single name.
- This is a composition – building complex structures from simple ones.
- Examples:

```
struct Point {
  int x;
  int y;
};
```

```
struct Employee {
  char firstName[20];
  char lastName[20];
  int age;
  double salaryWeight;
};
```

- Notice the ";" at the end of a structure type.

Structures

- The keyword `struct` defines a new data type.
- The variables declared within a structure are called *members*.
- Once defined, a structure variable can be declared like any other built-in data types:

```
struct Point pA, pB;
struct Point pC = {5, 10};
struct Employee e1, e2;
```

- Assignment operator copies every member of the structure (be careful with pointers).

Structures

- Members can be structures or self referential.
- Examples:

```
struct Triangle {
  struct Point pA;
  struct Point pB;
  struct Point pC;
};

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

Structures

- Individual members can be accessed using "." operator.

```
struct Point p = {5, 10};
int x = p.x;
int y = p.y;
```

- If structure is nested, multiple "." are required.

```
struct Rectangle {
  struct Point topLeft;
  struct Point bottomRight;
};
```

```
struct Rectangle r;
int x =  r.topLeft.x;
int y = r.topLeft.y;
```

Structure pointers

- Structures are copied element wise.
- If structures are large, it is more efficient to pass pointers:

```
void f(struct Point *pp);
struct Point p;
f(&p);
```

Structure pointers

- Members can be accessed from structure pointers using "->" operator.

```
struct Point p = {5, 10};
struct Point *pp = &p;
pp->x = 10;
int y = pp->y;
```

- Other ways to access structure pointers:

```
(*pp).x = 10;
int y = (*pp).y;
```

- Why is the () required?

Array of structures

- Declaring an array of integers: `int integerArray[10];`
- Declaring an array of points: `struct Point pointArray[10];`
- Initializing an array of integers:
  - `int integerArray[4] = {1, 2, 3, 4};`
- Initializing an array of points:
  - `struct Point pointArray[4] = {1,2,3,4,5,6,7,8};`
  - `struct Point pointArray[4] = {{1,2},{3,4},{5,6},{7,8}};`

Size of structures

- The size of a structure is *equal or greater than* the sum of the sizes of its members.
- This is an important issue in designing libraries, precompiled files and CPU processing instructions.

Content

Unions

- A union is a variable that may hold objects of different types/sizes in the same memory location.
- Example:

```
union Data {
  int iData;
  float fData;
  char *sData;
};

union Data d;
d.iData = 10;
d.fData = 3.14f;
d.sData = "Hello World";
```

Unions

- The size of a union variable is equal to the size of its largest element.
- Note that the compiler does not test if the data is being read in the correct format.

```
union Data d;
d.iData = 10;
float f = d.fData;
```

- What is the value of `f`?

Content

Review: dynamic memory allocation

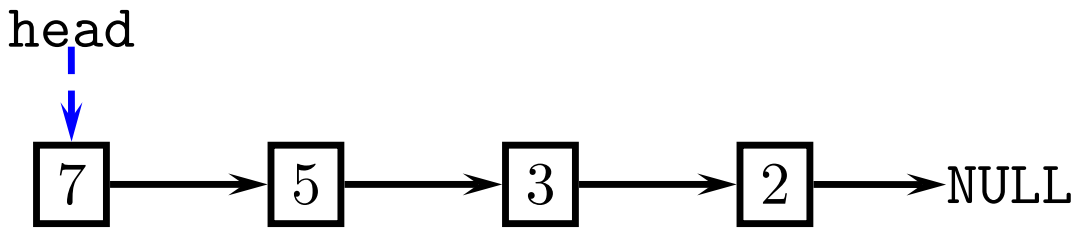
- `void* malloc(size_t n)`
  - `malloc()` allocates a block of memory
  - Returns a pointer to **uninitialized** block of memory on success, returns `NULL` on failure;
  - The returned value should be cast to appropriate type.
- Examples:
  - `int *ip = (int*)malloc(100*sizeof(int));`
  - `float *fp = (float*)malloc(50*sizeof(float));`

Review: dynamic memory allocation

- `void* calloc(size_t n, size_t size)`
  - Allocates an array of `n` elements, each of which is `size` bytes
  - Initializes memory to 0
- `void free(void*)`
  - Frees memory allocated by `malloc()` or `calloc()`.
  - Common error: accessing memory after calling `free()`.

Linked lists

- A **linked list** is a data structure which consists of a sequence of records where each element contains a link to the next record of the sequence.
- Linked lists can be *singly linked*, *doubly linked* or *circular*.
- Every node has a **datum** and a **link** to the next node in the list.
- The start of the list is called **head** which is maintained in a separate variable.
- End of the list is indicated by `NULL` (the sentinel).



Linked lists

- Declaration of a linked list containing integers:

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

struct Node *head = NULL;
```

- A comparison of linked lists and arrays:

	Linked lists	**Arrays**
Size	dynamic	fixed
Indexing	$O(n)$	$O(1)$
Inserting	$O(1)$	$O(n)$
Deleting	$O(1)$	$O(n)$

Create a new node for a linked list

Create a new element (node) given a datum:

```
struct Node* createElement(int datum) {
  struct Node* element = (struct Node*)malloc(
    sizeof(struct Node));
  if (element != NULL) {
    element->datum = datum;
    element->next = NULL;
  }
  return element;
}
```

Add an element to a linked list

Add an element to a linked list:

```
struct Node* addElement(struct Node *head, int datum) {
  struct Node *element = createElement(datum);
  if (element == NULL) {
    return head;
  } else {
    element->next = head;
    return element;
  }
}
```

If the element is not null, it becomes the new head of the list.

Create a linked list

Repeatedly create elements from data and add them to a list:

```
struct Node* createList(struct Node *head) {
  int datum;
  for (datum) {
    head = addElement(head, datum);
  }
  return head;
}
```

Note that the `for` loop depends on concrete data.

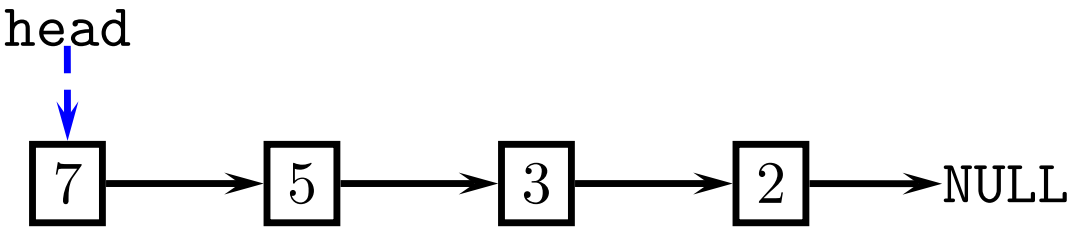
Browse a linked list

Use a loop to browse a linked list:

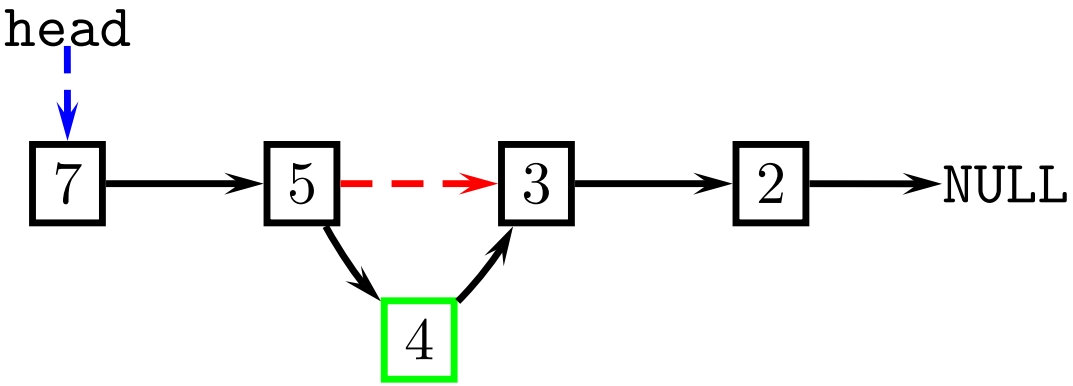
```
struct Node *p;
for (p = head; p != NULL; p = p->next) {
  /* do something */
}

for (p = head; p->next != NULL; p = p->next) {
  /* do something */
}
```

Insert an element to a linked list

Before insertion:

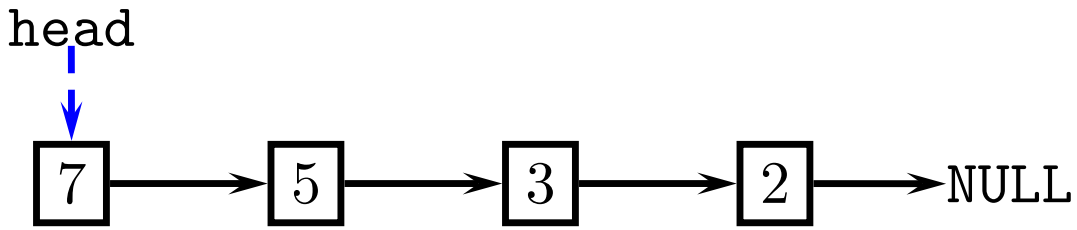


After insertion (red connection is removed):

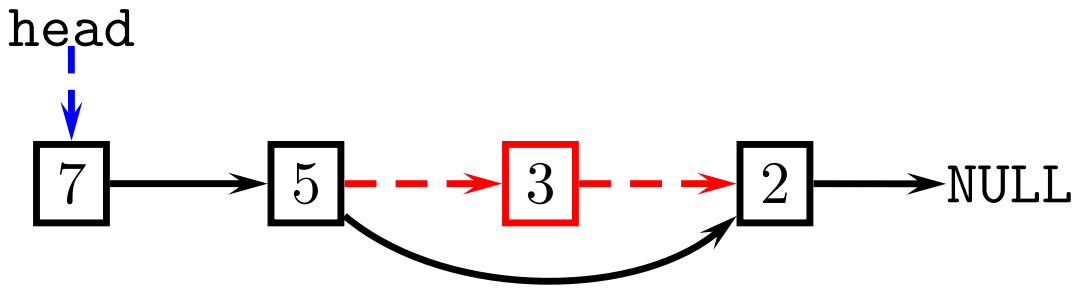


Remove an element from a linked list

Before removal:



After removal (red connections are removed):



Content

Content

Stacks

- A stack is a special type of list: last element in is first element out – LIFO.
  - Add an element to stack: `push()`
  - Remove an element from stack: `pop()`
- Read and write from the same end of list.
- Stacks can be implemented using linked lists or arrays.

Stack as array

- Store as array buffer, using either *static allocation* or *dynamic allocation* of memory.

```
int stackBuffer[100];
```

- Elements are added and removed from end of array. We need to track end:

```
int top = 0;
```

Stack as array

- Add element:

```
void push(int element) {
  stackBuffer[top++] = element;
}
```

- Remove element:

```
int pop() {
  if (top > 0) {
    return stackBuffer[top--];
  } else return 0;
}
```

Resizing array

- One drawback of implementing stack as array is that the user has to anticipate the maximum size of the array.
- What do we do if the maximum size cannot be provided in advance?
- Two solutions:
  - Resize the array
  - Choose a different implementation other than array

Resizing array

- How to grow array?
- The first try:
  - `push()`: increase the size of `stackBuffer` by 1.
  - `pop()`: decrease the size of `stackBuffer` by 1.
- However, this technique is too expensive:
  - We need to copy all elements to a new array;
  - If the array is full, inserting first n elements takes time proportional to

    $$1 + 2 + \cdots + n \approx n^2/2.$$

    This is infeasible for large n.
  - We need to ensure that array resizing happens infrequently.
- *Repeated doubling* technique: If array is full, create a new array of *twice* the size and copy elements.

Stack as linked list

- Store as linked list (dynamic allocation)

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

struct Node *stack = NULL;
```

- "Top" is now at the head of linked list, no need to track the top.

Stack as linked list

Add element:

```
void push(int element) {
  struct Node* node = (struct Node*)malloc(
    sizeof(struct Node));
  if (node != NULL) {
    node->element = element;
    node->next = stack;
    stack = node;
  }
}
```

Adding an element pushes back the rest of the stack.

Stack as linked list

Remove element:

```
int pop() {
  if (stack != NULL) {
    struct Node *p = stack;
    int element = p->element;
    stack = p->next;
    free(p);
    return element;
  } else {
    return 0;
  }
}
```

If the stack is null, we choose to return some special (default) value (for example, 0).

Content

Queues

- Opposite of stack: first in, first out – FIFO
  - Add an element to queue: `enqueue()`
  - Remove an element from queue: `dequeue()`
- Read and write from opposite ends of list
- Queues impose an ordering of elements, which is important for
  - User interface library (event queues, message queues)
  - Data package transmission (network)

Queue as array

- Store as array buffer, static or dynamic allocation

```
float queueBuffer[100];
```

- Elements are added to end (`rear`), removed from beginning (`front`).
- Need to keep track of `front` and `rear`:

```
int front = 0, rear = 0;
```

- Alternately, we can track the front and number of elements:

```
int front = 0, count = 0;
```

- We shall use the second way.

Queue as array

Add element:

```
void enqueue(float element) {
  if (count < 100) {
    queueBuffer[front + count] = element;
    count++;
  }
}
```

Remove element:

```
float dequeue() {
  if (count > 0) {
    count--;
    return queueBuffer[front++];
  } else {
    return 0; /* or a special value */
  }
}
```

Queue as array

- If we implement the queue this way, this would make a bad queue. Why?
- Consider a queue of capacity 4 where ↑ represents `front` and ↑ represents `rear`.

1.0	2.0	3.0	
↑			↑

- Enqueue 4.0 to the rear of the queue:

1.0	2.0	3.0	4.0
↑			↑

The queue is now full.

Queue as array

- If we implement the queue this way, this would make a bad queue. Why?
- Consider a queue of capacity 4 where ↑ represents `front` and ↑ represents `rear`.

1.0	2.0	3.0	
↑			↑

- Enqueue 4.0 to the rear of the queue:

1.0	2.0	3.0	4.0
↑			↑

The queue is now full.

Queue as array

- If we implement the queue this way, this would make a bad queue. Why?
- Consider a queue of capacity 4 where ↑ represents `front` and ↑ represents `rear`.

1.0	2.0	3.0	
↑			↑

- Enqueue 4.0 to the rear of the queue:

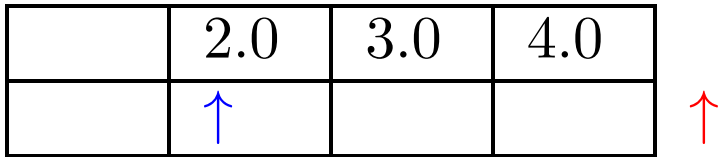
1.0	2.0	3.0	4.0
↑			

↑

The queue is now full.

Queue as array

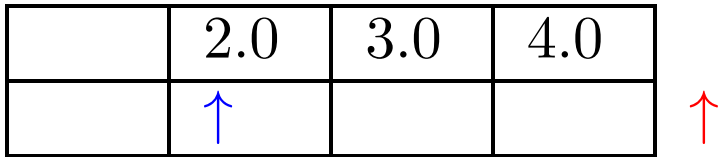
- Dequeue 1.0:



- Enqueue 5.0 to the rear, where should it go?
- Solution: use a *circular buffer*: 5.0 should go in the beginning of the array.

Queue as array

- Dequeue 1.0:



- Enqueue 5.0 to the rear, where should it go?
- Solution: use a *circular buffer*: 5.0 should go in the beginning of the array.

Queue as array

Modified `enqueue()`:

```
void enqueue(float element) {
  if (count < 100) {
    queueBuffer[(front+count) % 100] = element;
    count++;
  }
}
```

Queue as array

Modified `dequeue()`:

```
float dequeue() {
  if (count > 0) {
    float element = queueBuffer[front];
    count--;
    front++;
    if (front == 100) {
      front = 0;
    }
    return element;
  } else {
    return 0; /* or a special value */
  }
}
```

Why it would be harder if we use "front" and "rear" counters?

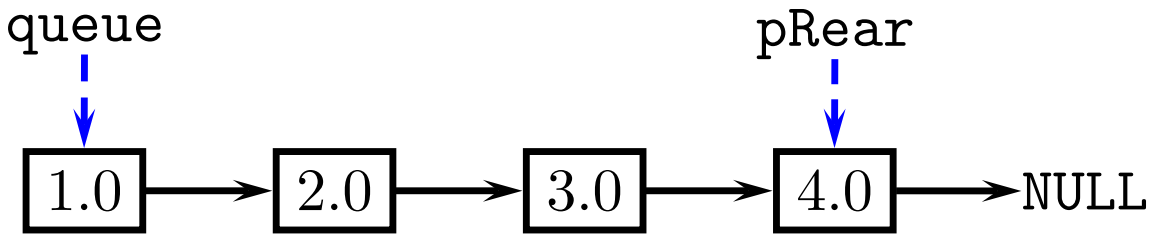
Queue as linked list

- Store as linked list:

```
struct Node {
  float element;
  struct Node *next;
};

struct Node *queue = NULL;
```

- Let front be at beginning, no need to track front.
- Rear is at end, we should track it:

```
struct Node *pRear = NULL;
```



Queue as linked list – Add element

Add element:

```
void enqueue(float element) {
  struct Node *node = (struct Node*)malloc(
      sizeof(struct Node));
  node->element = element;
  node->next = NULL;
  if (pRear != NULL) {
    pRear->next = node;
  } else {
    queue = node;
  }
  pRear = node;
}
```

Adding element does not affect the front if the queue is not empty.

Queue as linked list – Remove element

Remove element:

```
float dequeue() {
  if (queue != NULL) {
    struct Node *pElement = queue;
    float element = pElement->element;
    queue = pElement->next;
    if (pElement == pRear) {
      pRear = NULL;
    }
    free(pElement);
    return element;
  } else {
    return 0; /* or a special value */
  }
}
```

Removing element does not affect the front if the queue is not empty.

Content

Exercises

Exercise 1. Implement a linked list of prime numbers which are less than or equal a given positive integer n.

- create the list;
- print the list to verify its content;
- reverse the list;
- free the list.

Exercise 2. Implement a list of n student names

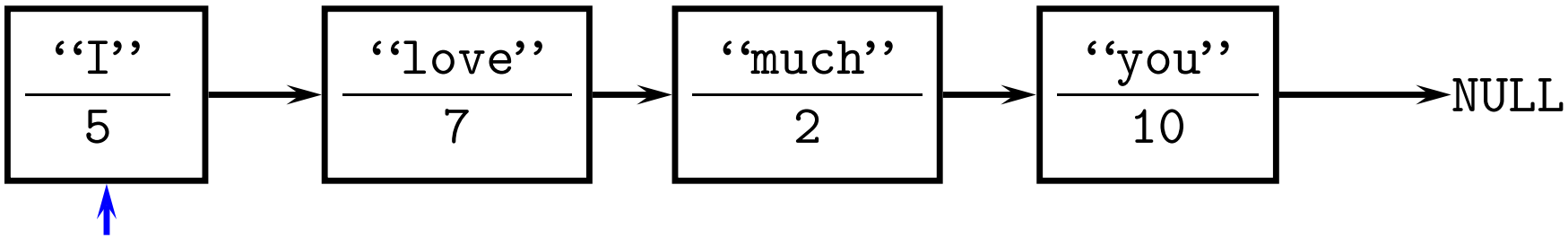
- create the list, names are read from a text file;
- sort the list in lexicographical order using a sorting algorithm;
- scan a name from keyboard, insert it to the list at correct position;
- search for a given name, return its first positions if success;
- delete all nodes whose name matches a given name;
- free the list.

Exercises

Exercise 3. We need to count the words of an English text file using a linked list.

- Input: an English text file
- Output: a sorted linked list, each node of the list contains an English word and its frequency

Note that the number of nodes is the number of different words appeared in the input text. The list is sorted alphabetically.



Exercises

Exercise 4. Implement stacks and queues using two data structures (array or linked list).

Exercise 5. (optional) Design your stack and queue implementations so that they can be reused for different data types (integers, fractions, strings, user-defined data types...)