Linkedlist Operations (Singly and Doubly)

Max Le

Monday 23-08-2021

Contents

1 Theory

1.1 Array vs Linkedlist

1.1.1 Array

1. fixed size
2. create on stack e.g. int A[5]
3. create on heap e.g. int *p = new int[5];

1.1.2 Linkedlist

1. Allocate memories in node: pointer to current elemen | pointer to next element
2. Create on heap

1.2 LinkedList basic

1.2.1 collections of nodes, each node has "Data" + "Pointer to next Node"

1.2.2 "Head" points to first node

1.2.3 Addresses may not be side by side, unlike array

1.2.4 To create a node, need: DATA and POINTER to a NODE (NODE type)

1.2.5 A node structure: DATA | NEXT

1.2.6 For C language:

```
struct Node
{
  int data;
  struct Node *next;                /* self referential
   ↪  structure */
};

int main(){
  struct Node *p;
  p = (struct Node*)malloc(sizeof(struct Node));
  /* For C++ */
  /* p = new Node */


  p->data = 10;                  /* assign 10 to DATA */
  p->next = 0;                         /* or NULL, meaning
   ↪  next pointer points to nothing */

  return 0;
}
```

1.2.7 How to display a linkedlist

1. Iterative

```c
struct Node *p = first;              /* set a pointer
↪  to point to first node */

while (p != 0) {
  printf("%d\n", p->data);         /* print the data */
  p = p ->next;                         /* move p to next
   ↪  node */
 }
```

2. Recursive

```c
void displayRecursive(struct Node *p){
  if (p != NULL) {
    printf("%d\n", p->data);
    displayRecursive(p->next);
  }
}
```

1.3 Singular Linked List

1.3.1 How to count node in linked list

1. Iterative

```c
/* taking O(n) time and O(1) space */
int count(struct Node *p){
  int c = 0;
  while (p != NULL) {
    c++;
    p = p->next
      }
  return c;
}
```

2. Recursive

```c
/* Time O(n) Space O(n) */
int count (struct Node *p){
  if (p == NULL) {
    return 0;
```

```
  }
  else {
    return count(p->next) + 1;
  }
}
```

1.3.2 How to sum all elements in linked list

1. Iterative

```
/* Time O(n) Space O(1)*/
int Add(struct Node *p){
  int sum = 0;
  while (p != NULL) {
    sum = sum + p->data;
    p=p->next;
  }
  return sum;
}
```

2. Recursion

```
/* Time, Space O(n) */
int Add(struct Node *p){
  if (p == NULL) {
    return 0;
  }
  else {
    return Add(p->next)+p->data;
  }
}
```

1.3.3 Find max element in linked list

1. Iterative

```
int max(struct Node *p){
  int m = -32768;                /* min integer */
  while (p != NULL) {
    if (p->data > m) {
```

```
      m = p->data;
    }
    p = p->next;
  }
  return m;
}
```

2. Recursive

```
int max(struct Node *p){

  int x = 0;

  if (p == NULL) {
    return MIN_INT;
  }
  else {
    x = max(p->next);
    if (x > p->data) {
      return x;
    }
    else {
      return p->data;
    }
  }
}
```

```
int max(struct Node *p){
  int x = 0;
  if (p == 0) {
    return MIN_INT;
  }
  x = max(p->next);
  return x > p->data ? x : p->data;
}
```

1.3.4 Searching (linear search)

1. Iterative

```c
Node *search (struct Node *p, int key){
  while (p != NULL) {
    if (key == p->data) {
      return(p);
    }
    p = p->next;
  }
  return NULL;
}
```

2. Recursive

```c
Node *search(struct Node *p, int key){
  if (p == NULL) {
    return NULL;
  }

  if (key == p->data) {
    return p;
  }

  return search(p->next, key);
}
```

3. Move found to head

```c
Node *search(struct Node *p, int key){
  Node *q = NULL;                    /* previous pointer */

  while (p != NULL) {
    if (key == p->data) {
      q->next = p->next;
      p->next = first;
      first = p;
    }
    q = p;
    p = p->next;
  }
}
```

1.3.5 Inserting

1. Insert **BEFORE** first node

```
/* constant time */
Node *t = new Node;                 /* create new node */
t->data = x;                        /* assign new node
↪  data */
t->next = first;                  /* t points to first
↪  pointer, making t comes before first */
first = t;                          /* old "first" point
↪  to t , t is now first */
```

2. Insert **AFTER** given position

```
/* insert between left and right node */
/* O(N) max time, O(1) min time */
Node *t = new Node;
t->data = x;
p = first;                          /* start temporary
↪  pointer from first */
pos = 4;                          /* position to insert
↪  after */

/* moving p till reach left node */
for (i = 0; i < pos-1 ; i++) {
  p = p->next;
 }

t->next = p->next;                  /* t next pointer
↪  points to the right node */
p->next = t;                          /* p->next points
↪  to t, so t is between left and right */
```

3. Combine

```
void Insert (int pos, int x){
  Node *t, *p;
  if (pos == 0) {
    t = new Node;
    t->data = x;
```

```
    t->next = first;
    first = t;
  }
  else if (pos > 0) {
    p = first;
    for (i = 0; i < pos-1 && p != NULL ; i++) {
      p = p->next;
    }

    if (p != NULL) {
      t = new Node;
      t->data = x;
      t->next = p->next;
      p->next = t;
    }
  }
}
```

4. Special case: Insert at last only

```
void InsertLast(int x){
  Node *t = new Node;
  t->data = x;
  t->next = NULL;

  /* no node in list */
  if (first == NULL) {
    first = last = t;
  }
  else {
    last->next = t;
    last = t;
  }
}
```

5. Insert in a **SORTED** linked list, at a **SORTED** position

```
/* Time: min O(1) max O(n) */
p = first;
q = NULL;
```

```
while (p != NULL && p->data < x) {
  q = p;
  p = p->next;
 }

t = new Node;
t->data = x;
t->next = q->next;
q->next = t;
```

1.3.6 Deleting

1. Delete first node

```
/* Time O(1)  */
Node *p = first;                /* arbitray pointer p
↪  poins to first */
first=first->next;               /* move first to point
↪  to next node */
delete p;                        /* delete the original
↪  first */
```

2. Delete at given position

```
/* Time min O(1) max O(n) */
Node *p = first;
Node *q = NULL;

for (i = 0; i < pos-1 ; i++) {
  q = p;
  p = p->next;
 }

q->next = p->next;
delete p;
```

1.3.7 Check if linkedlist is sorted

```
/* Time O(n) max O(1) min */
int x = -32768;
Node *p = first;

while (p != NULL) {
  if (p->data < x) {
    return false;
  }
  x = p->data;
  p = p->next;
 }
return true;
```

1.3.8 Remove duplicate

```
Node *p = first;
Node *q = first->next;
while (q != NULL) {
  if (p->data != q->data) {
    p = q;
    q = q->next;
  }
  else {
    p->next = q->next;
    delete q;
    q = p->next;
  }
 }
```

1.3.9 Reverse a linkedlist

1. Interchange elements

```
p = first;
i = 0;
/* Copy to extra array */
while (p != NULL) {
  A[i] = p->data;
```

```
  p = p->next;
  i++;
 }
p = first;
i--;

/* Reverse copy back to list */
while (p != NULL) {
  p->data = A[i];
  i--;
  p = p->next;
 }
```

2. Reversing links

```
/* Setup 3 sliding pointers */
p = first;
q = NULL;
r = NULL;

while (p != NULL) {
  r = q;
  q = p;
  p = p->next;

  q->next = r;
 }

/* Update first */
first = q;
```

3. Recursion

```
void Reverse(Node *q, Node *p){
  if (p != NULL) {
    Reverse(p, p->next);
    p->next = q;
  }
  else {
    first = q;
```

```
  }
}
```

1.3.10 Joining/Append 2 linked list

```
p = first;

/* traverse till the last node and stop */
while (p->next != NULL) {
  p = p->next;
 }

p->next = second;                   /* point last node to first
↪  node of the other list */
second = NULL;                          /* delete/free/NULL
↪  the extra pointer */
```

1.3.11 Merging 2 linkedlist

```
/* Create 2 pointers for the merged list */
Node *third, *last;

/* First loop */
if (first->data < second->data) {
  third = last = first;
  first = first->next;
  last->next = NULL;
 }
 else {
   third=last=second;
   second = second->next;
   last->next = NULL;
 }

while (first != NULL && second != NULL) {
  if (first->data < second->data) {
    last->next = first;
    last = first;
```

```
    first = first->next;
    last->next = NULL;
  }
  else {
    last->next = second;
    last = second;
    second = second->next;
    last->next = NULL;
  }
 }

if (first != NULL) {
  last->next = first;
 }
 else {
   last->next = second;
 }
```

1.3.12 Check for LOOP in

1. LOOP: Last node points to some other nodes
2. LINEAR: Last node points to NULL

   (a)
   ```
   int isLoop(Node *first){
     Node *p, *q;
     p = q = first;
     do
       {
         p = p-next;
         q = q->next;
         if (q != NULL) {
           q = q->next;
         }
         else {
           q = NULL;
         }
       } while (p != NULL && q != NULL);
   ```

```
  if (p == q) {
    return true;
  }
  else {
    return false;
  }


}
```

1.3.13 Circular linkedlist

1. Last node points to first node
2. or a collection of nodes that are circularly connected
3. Use HEAD instead of FIRST
4. Two representations:
   (a) HEAD/1st node -> 2nd node -> 3rd node -> 4th node -> back to HEAD/1st node
   (b) HEAD -> 1st -> 2nd -> 3rd -> 4th -> back to 1st
5. How to display
   (a) Loop display

   ```
   void Display(Node *p){
     do
       {
         printf("%d\n", p->data);
         p = p->next;
       } while (p != Head);

   }
   ```

   (b) Recursive display

   ```
   void Display(Node *p){
     static int flag = 0;                   /* so only 1
     ↪  creation of int flag */
     if (p != Head || flag = 0) {
   ```

```
        flag = 1;
        printf("%d\n", p->data);
        Display(p->next);
    }
    flag = 0;
}
```

6. How to insert

   (a) After Head

```
Node *t;
Node *p = Head;

for (i = 0; i < pos-1 ; i++) {
    p = p->next;
 }

t = new Node;
t->data = x;
t->next = p->next;
p->next = t;
```

   (b) Before Head

```
Node *t = new Node;
t->data = x;
t->next = Head;

Node *p = Head;
while (p->next != Head) {
    p = p->next;
 }

p->next = t;
Head = t;
```

7. Delete

   (a) Delete from Given Position

```
p = Head;
for (i = 0; i < pos-2 ; i++) {
  p = p->next;
 }

q = p->next;
p->next = q->next;
x = q->data;
delete q;
```

(b) Delete Head

```
p = Head;
while (p->next != Head) {
  p = p->next;
 }

p->next = Head->next;
x = Head->data;
delete Head;
Head = p->next;
```

1.4 Doubly Linked List

1.4.1 a node has pointer to NEXT node and PREVIOUS node

1.4.2 Structure in C:

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

1.4.3 Insertion

1. Before first

```
Node *t = new Node;
t->data = x;
```

```
/* Modify links */
t->prev = NULL;
t->next = first;
first->prev = t;

/* Rename new first */
first = t;
```

2. After given index min O(1) max O(n)

```
Node *t = new Node;
t->data = x;

/* To reach the node before the insertion */
for (i = 0; i < pos-1 ; i++) {
  p = p->next
    }

/* Modify links */
t->next = p->next;                /* inserted node
↪  should point to the node on the RIGHT (p->next) */
t->prev = p;                        /* inserted node
↪  should point to the node on the LEFT (p) */


/* for node on the RIGHT, its prev must point to the
↪  inserted node */
/* Must check if next is available, in case we insert at
↪  last then next is NULL */
if (p->next ! =NULL) {
  p->next->prev = t;
 }

p->next = t;                         /* node on the LEFT
↪  should point to the inserted node */
```

1.4.4 Delete

1. Delete 1st node

```
p = first;
first = first->next;
x = p->data;
delete p;

if (first != NULL) {
  first->prev = NULL;
 }
```

2. Delete from given index

```
/* bring a pointer p upon given index */
p = first;

for (i = 0; i < pos-1 ; i++) {
  p = p->next;
 }

p->prev->next = p->next;         /* LEFT node points to
↪  RIGHT node, skip CURRENT */

if (p->next != NULL) {                /* if RIGHT node
↪  exists */
  p->next->prev = p->prev;          /* RIGHT node points
  ↪  to LEFT node, skip CURRENT */
 }

x = p->data;
delete p;
```

1.4.5 Reverse

1. Display

```
p = first;

while (p != NULL) {
  printf("%d\n", p->data);
  p = p->next
```

```
  }
```

2. Reverse

```
p = first;

while (p != NULL) {
  temp = p->next;
  p->next = p->prev;
  p->prev = temp;
  p = p->prev;
 }

/* for last node, bring first there */
if (p!= NULL && p->next == NULL) {
  first = p;
 }
```

1.4.6 Circular

1. Insert
2. Display

2 Comparing different linkedlists

2.1 Space: Doubly takes double the amount of pointers in Singly

2.2 Insert: constant time for Linear Singly, Linear Doubly, Circular Doubly. Nth time for Circular Singly

3 Array vs Linkedlist

3.1 Creation: array in stack or heap, LL is always on heap

3.2 Size: array fixed, LL can grow until heap is full

3.3 LL takes extra space

3.4 Array access directly (faster), LL access sequentially (slower)

3.5 Data movement in array is more expensive

4 Student challenge

4.1 Find middle node of LL

4.1.1 1st solution

1. find length of LL => say 7
2. reach middle node => 7/2 ~ 4
3. move the pointer 4-1 times

4.1.2 2nd solution

1. use 2 pointers, 1 move 2 space, other move 1 space

4.1.3 3rd solution

1. push each node to a stack
2. pop out the node at stack number floor(stack size / 2)

4.2 Find intersection of 2 LL

4.2.1 Eg: LL1->LL3 and LL2->LL3 => Both LL1 and LL2 has LL3 as common. For example:

1. LL1: 8->6->3->9->10->4->2->12
2. LL2: 20->30->40->10->4->2->12

4.2.2 We need to find the starting common point, i.e. block 10?

1. We traverse the 1st LL or 2nd LL till the end.
2. Then from end, we traverse back, if the previous block is different,
3. then current block is **intersection point**
4. But cannot traverse back in Single LL?? Use Stack
   (a) Traverse each LL, store address in a stack

(b) Then compare two stacks, whenever the address differ, that location is the **intersection**

5. Suppose our LL1 and LL2 has the following address

   (a) LL1: length 8

Data	Address
8	100
6	110
3	130
9	150
10	**200**
4	220
2	240
12	260

   (b) LL2: length 7

Data	Address
20	300
30	310
40	330
10	**200**
4	220
2	240
12	260

   (c) Record address in a stack. **Note**: first address goes to bottom:

Stack 1	Stack 2
260	260
240	240
220	220
200	**200**
150	330
130	310
110	300
100	

      i. Comparing the two stacks, pops out an address, if same, delete

      ii. Keep track of the previous address that we pop out.

Stack 1	Stack 2
260	260
240	240
220	220
200	200
150	**330**
130	310
110	300
100	

iii. Addresses **150** and **330** are different => The intersection is the node before

iv. i,e, node with address 200, which contains data **10**

4.3 Sparse matrix using LL

4.3.1 Suppose we have the following matrix 5 x 6

.	/	**0**	**1**	**2**	**3**	**4**	**5**
0	/	0	0	0	0	8	0
1	/	0	0	0	7	0	0
2	/	5	0	0	0	9	0
3	/	0	0	0	0	0	3
4	/	6	0	0	4	0	0

4.3.2 We want to AVOID STORING ZEROES

4.3.3 Coordinate, for number 7 is at (1,3) 1st row, 3rd column

4.3.4 For each row, we represent the non zero as an LL. Eg:

1. **Row 0**: |-4-|-8-|-NULL-|

   (a) Row 0 has element at 4th column, data is 8, it has no other, so pointer next is NULL

2. **Row 1**: |-3-|-7-|-NULL-|

3. **Row 2**: |-0-|-5-|——| –> |-4-|-9-|-NULL-|

   (a) Here Row 2 at 0th column has element 5, there is also element 9 at 4th column, so **next** point to this.

4. **Row 3**: |-5-|-3-|-NULL-|

5. **Row 4**: |-0-|-6-|——| –> |-3-|-4-|-NULL-|

4.3.5 This is called an ARRAY OF LINKED LIST

4.3.6 So, the Node structure contains: Column, Value, Next

4.3.7 To make this structure in C:

```
struct Node
{
  int col;
  int val;
  struct Node *next;
};
```

>

4.3.8 We need the number of row, so we can create an array of size row

4.3.9 In our example, we have 5 rows x 6 cols. Generalized to m x n matrix, where m = 5

```
/* Create array of size m */
Node *A[m];

/* For each row, we create a new node */
A[0] = new Node;
```

>

4.4 Polynomial representations

4.4.1 Consider: $P(x) = 4x^3 + 9x^2 + 6x + 7$

4.4.2 We will construct each term as a node that has: coefficient, exponent and next pointer

```
struct Node
{
  int coeff;
  int exp;
  struct Node *next;
};
```

5 Codes

5.1 Implement in C: `linkedlistBasic.c`

5.2 Implement in C++: `linkedlistBasic.cpp`

5.3 Circular linkedlist in C: `circularLinkedList.c`

5.4 Circular Doubly LinkedList in C: `circularDoubly.c`

5.5 Doubly linkedlist in C: `doubleLinkedList.c`

5.6 Student challenge intersection: `challengeIntersection.c`

5.7 Student challenge middle node: findMiddle

5.8 Student challenge sparse matrix: sparseMatrix

5.9 Basic Stack implementation: stackBasic