CS 2412 Data Structures

Chapter 4

Lists

Definition

A list is an ordered collection of data in which an addition or deletion can be performed at any position.

In general a list is neither FIFO nor FILO. The entries can be in and out in random.

Basic operations:

- Operations creation, destroy, empty, full, size are similar to the queue.
- Insertion: to determine the position where the data will insert, a key is used to identify a node.
- Deletion: after finding the location of the data, the node is deleted from the list.
- Retrieval: after finding the location of the data, the data is retrieved, but the list is unchanged.
- Traversal: processes each element in a list in sequence.

We will use pseudo codes to describe these functions.

ALGORITHM 5-2 Insert List Node

```
Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.
  Pre    list is metadata structure to a valid list
         pPre is pointer to data's logical predecessor
         dataIn contains data to be inserted
  Post   data have been inserted in sequence
  Return true if successful, false if memory overflow
 1 allocate (pNew)
 2 set pNew data to dataIn
 3 if (pPre null)
   Adding before first node or to empty list.
   1  set pNew link to list head
   2  set list head to pNew
 4 else
   Adding in middle or at end.
   1  set pNew link to pPre link
   2  set pPre link to pNew
 5 end if
 6 return true
end insertNode
```

We need to allocate the position by key. Also we need to distinct that the insertion happens at the list head or other positions.

ALGORITHM 5-3 List Delete Node

```
Algorithm deleteNode (list, pPre, pLoc, dataOut)
Deletes data from list & returns it to calling module.
  Pre  list is metadata structure to a valid list
       Pre is a pointer to predecessor node
       pLoc is a pointer to node to be deleted
       dataOut is variable to receive deleted data
  Post data have been deleted and returned to caller
 1 move pLoc data to dataOut
 2 if (pPre null)
   Deleting first node
   1  set list head to pLoc link
 3 else
   Deleting other nodes
   1  set pPre link to pLoc link
 4 end if
 5 recycle (pLoc)
end deleteNode
```

Similar to insertion, we need to consider if the head of the list will be deleted.

We need a search function to find out the position of certain data.

ALGORITHM 5-4 Search List

```
Algorithm searchList (list, pPre, pLoc, target)
Searches list and passes back address of node containing
target and its logical predecessor.
  Pre   list is metadata structure to a valid list
        pPre is pointer variable for predecessor
        pLoc is pointer variable for current node
        target is the key being sought
  Post  pLoc points to first node with equal/greater key
        -or- null if target > key of last node
        pPre points to largest node smaller than key
        -or- null if target < key of first node
  Return true if found, false if not found
1 set pPre to null
2 set pLoc to list head
3 loop (pLoc not null AND target > pLoc key)
  1  set pPre to pLoc
  2  set pLoc to pLoc link
4 end loop
5 if (pLoc null)
  Set return value
  1  set found to false
```

continued

ALGORITHM 5-4 Search List *(continued)*

```
6 else
  1  if (target equal pLoc key)
     1  set found to true
  2  else
     1  set found to false
  3  end if
7 end if
8 return found
end searchList
```

The above search list function is for searching an "ordered list", where the keys are arranged in some order. And the search begins always from the list head. The search function can be a little complicated if the list is not ordered.

We could make the search slightly more efficient if we have the rear pointer of the list. We can test to see if the searched key is greater than the last key.

Dynamic liked list

```
// List ADT Type Defintions
typedef struct node
   {
    void*         dataPtr;
    struct node* link; //pointer to next node
   } NODE;
typedef struct
   {
    int   count;
    NODE* pos; // position determined by application
    NODE* head;
    NODE* rear;
    int    (*compare) (void* argu1, void* argu2);
   } LIST;
```

```
LIST* createList
     (int (*compare) (void* argu1, void* argu2))
{
LIST* list;
list = (LIST*) malloc (sizeof (LIST));
if (list)
   {
    list->head    = NULL;
    list->pos     = NULL;
    list->rear    = NULL;
    list->count   = 0;
    list->compare = compare;
   } // if
return list;
} // createList
```

```
/*    Pre    pList is pointer to valid list
           dataInPtr pointer to insertion data
   Post   data inserted or error
   Return -1 if overflow
           0 if successful
           1 if dupe key  */
int addNode (LIST* pList, void* dataInPtr)
{
bool found;
bool success;
NODE* pPre;
NODE* pLoc;
found = _search (pList, &pPre, &pLoc, dataInPtr);
if (found)  // Duplicate keys not allowed
   return (+1);
success = _insert (pList, pPre, dataInPtr);
if (!success)  // Overflow
   return (-1);
return (0);
} // addNode
```

```
/*    Pre    pList pointer to a valid list
           pPre  pointer to data's predecessor
           dataInPtr data pointer to be inserted
      Post   data have been inserted in sequence
      Return boolean, true  if successful,
                      false if memory overflow
*/
static bool _insert (LIST* pList, NODE* pPre,
                     void* dataInPtr)
{
NODE* pNew;
if (!(pNew = (NODE*) malloc(sizeof(NODE))))
   return false;
pNew->dataPtr   = dataInPtr;
pNew->link      = NULL;
if (pPre == NULL)
   {             // Adding before first node or to empty list.
    pNew->link       = pList->head;
    pList->head      = pNew;
    if (pList->count == 0)   // Adding to empty list. Set rear
       pList->rear = pNew;
   } // if pPre
else
   {          // Adding in middle or at end
    pNew->link  = pPre->link;
    pPre->link  = pNew;        // Now check for add at end of list
    if (pNew->link     == NULL)
         pList->rear   =  pNew;
   } // if else
(pList->count)++;
return true;
} // _insert
```

```
/*    Post   Node deleted or error returned.
   Return false not found; true deleted
*/
bool removeNode  (LIST*  pList, void*  keyPtr,
                  void** dataOutPtr)
{
bool found;
NODE* pPre;
NODE* pLoc;
found = _search (pList, &pPre, &pLoc, keyPtr);
if (found)
   _delete (pList, pPre, pLoc, dataOutPtr);
return found;
} // removeNode
```

```
/* ================= _delete ================
Deletes data from a list and returns
pointer to data to calling module.
   Pre    pList pointer to valid list.
          pPre  pointer to predecessor node
          pLoc  pointer to target node
          dataOutPtr pointer to data pointer
   Post   Data have been deleted and returned
          Data memory has been freed
*/
void _delete (LIST* pList, NODE*  pPre,
              NODE* pLoc,  void** dataOutPtr)
{
// Statements
*dataOutPtr = pLoc->dataPtr;
if (pPre == NULL)
   // Deleting first node
   pList->head = pLoc->link;
else
   // Deleting any other node
   pPre->link = pLoc->link;

// Test for deleting last node
if (pLoc->link == NULL)
    pList->rear = pPre;

(pList->count)--;
free (pLoc);

return;
} // _delete
```

```
bool searchList (LIST*  pList, void* pArgu,
                 void** pDataOut)
{
bool  found;
NODE* pPre;
NODE* pLoc;
found = _search (pList, &pPre, &pLoc, pArgu);
if (found)
    *pDataOut = pLoc->dataPtr;
else
    *pDataOut = NULL;
return found;
} // searchList
```

```
bool _search (LIST*  pList, NODE** pPre, //pointer to predecessor
              NODE** pLoc, /*point to receive node*/
               void*  pArgu /* key being sought*/)
{  //Macro Definition
#define COMPARE \
( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
#define COMPARE_LAST \
((* pList->compare) (pArgu, pList->rear->dataPtr))

int result; //local definition
*pPre  = NULL;
*pLoc  = pList->head;
if (pList->count == 0)
    return false;
if ( COMPARE_LAST > 0)
   {
    *pPre = pList->rear;
    *pLoc = NULL;
    return false;
   } // if
while ( (result = COMPARE) > 0 )
```

```
   {   // Have not found search argument location
         *pPre = *pLoc;
    *pLoc = (*pLoc)->link;
   } // while
if (result == 0)  // argument found--success
   return true;
else
   return false;
} // _search
```

```
/* This algorithm retrieves data in the list without
changing the list contents.  */
static bool retrieveNode (LIST*  pList ,void*  pArgu,
                                  void** dataOutPtr)
{
bool  found;
NODE* pPre;
NODE* pLoc;
found = _search (pList, &pPre, &pLoc, pArgu);
if (found)
   {
    *dataOutPtr = pLoc->dataPtr;
    return true;
   } // if
*dataOutPtr = NULL;
return false;
} // retrieveNode
```

```
bool traverse (LIST*  pList,
               int    fromWhere,
               void** dataPtrOut)
{
if (pList->count == 0)
    return false;
if (fromWhere == 0)
   {
    pList->pos  = pList->head;
    *dataPtrOut = pList->pos->dataPtr;
    return true;
   } // if fromwhere
else
   {
    if (pList->pos->link == NULL)
        return false;
```

```
    else
       {
        pList->pos  = pList->pos->link;
        *dataPtrOut = pList->pos->dataPtr;
        return true;
       } // if else
   } // if fromwhere else
} // traverse
```

```
LIST* destroyList (LIST* pList)
{
NODE* deletePtr;
if (pList)
   {
    while (pList->count > 0)
       {
        free (pList->head->dataPtr);   // delete data
          deletePtr    = pList->head;
        pList->head  = pList->head->link;
        pList->count--;
        free (deletePtr);   //  delete node
       } // while
    free (pList);
   } // if
return NULL;
} // destroyList
```

Comparison of implementations

Contiguous storage is generally preferable: when the records are individually vary small; when the size of the list is known when the program is written; when few insertions or deletions need to be made except at the end of the list; and when random access is important.

Linked storage proves superior: when the records are large; when the size of the list is not know in advance; and when flexibility is needed in inserting, deleting and rearranging the entries.

Linked lists in Arrays

Using arrays for linked lists have applications in certain cases.

- the number of entries in a list is known in advance.
- the links are frequently rearranged, but relatively few insertions or deletions are made.
- the same data are sometimes best treated as a linked list and other times as a contiguous list.

Example: sort a list of names.

Implementation

If a list is implemented by contiguous memory (array), then `InsertList` and `DeleteList` need to move entries backward or forward. So the amount of work the function does is approximatively *proportional* to n, where n is the length of the list.

Note: In processing a contiguous list with n entries:

`InsertList` and `DeletList` require time approximately proportional to n. `CreateList, ClearList, ListEmpty, ListFull, ListSize, ReplaceList` and `RetrieveList` operate in constant time.

Operations

The operations `CreateList`, `ClearList`, `ListEmpty`, `ListFull`, `ListSize` are similar to those of stacks and queues. The other operations can be defined as follows.

```
void InsertList(Position p, ListEntry x, List *list);
```

Pre: The list `list` has been created, `list` is not full, `x` is a valid list entry, and $0 \leq$ `p` $\leq n$, where n is the number of entries in list.

Post: `x` has been inserted into position `p` in `list`; the entry formerly in position `p` (provided `p` $< n$) and all later entries have their position number increased by 1.

```
void RemoveList(Position p,ListEntry *x, List *list);
```

Pre: The list `list` has been created, `list` is not empty, and $0 \leq$ `p` $< n$, where n is the number of entries in list.

Post: The entry in position `p` of `list` has been returned as `x` and deleted from `list`; the entries in all later positions (provided `p` $< n - 1$) have their position numbers decreased by 1.

Other operations:

`RetrieveList(Position p, ListEntry *x, List *list);`

Pre: The list `list` has been created, `list` is not empty, and $0 \leq$ `p` $< n$, where n is the number of entries in list.

Post: The entry in position `p` of `list` has been returned as `x`. `list` remains unchanged.

`ReplaceList(Position p, ListEntry x, List *list);`

Pre: The list `list` has been created, `list` is not empty, `x` is a balid list entry, and $0 \leq$ `p` $< n$, where n is the number of entries in list.

Post: The entry in position `p` of `list` has been replaced by `x`. The other entries of `list` remain unchanged.

`bool Traverse(List *list, Position p, void ** dataPr)`

Pre: The `list` has been created, `list` is not empty, and $0 \leq$ `p` $< n$, where n is the number of entries in list.

Post: The entry in position `p` of `list` has been returned to `*dataPr`. The position moves to next entry and return `true` if `p < n-1`, or return `false` if `p = n-1`.

Storage classes in C

In C, there are several storage classes of variables, which provide information about their visibility, lifetime and location.

`auto, register, static, extern`

The external declaration tells the compiler the type of the variable and that the compiler should assume space for it is allocated elsewhere. The declaration leaves the linker to resolve the references.

Example: `extern ListNode workspace[];`

```
typedef int ListIndex;

typedef struct listnode{
  ListEntry entry;
  ListIndex next;
} ListNode;

typedef struct list{
  ListIndex head;
  int count;
} List;

extern ListIndex avail, lastused;
extern ListNode workspace[];
```

Array `workspace[]` is the reserved space for all the list nodes.

An index `lastused` indicates the position of the last node that has been used at some time. Location with indices greater that `lastused` have never been allocated.

An index `avail` is used to denote the top of a stack that records the nodes have been used and then returned to available space. Note that the stack is also using the `workspace`.

Example: If we have a `workspace[5]`. We want to do the following operations:

Insert `A,B,C,D`, delete `C`, insert `E`, delete `A,B`, insert `C, B`.

initial: `list.head = -1, list.count = 0, avail=-1, lastused=-1`

insert A: `workspace[0].entry=A, workspace[0].next=-1, list.head=0, list.count=1, lastused=0, avial=-1`

insert B: `workspace[1].entry=B, workspace[1].next=-1, workspace[0].next=1, list.count=2, lastused=1`

insert C: `workspace[2].entry=C, workspace[2].next=-1, workspace[1].next=2, list.count=3, lastused=2`

insert D: `workspace[3].entry=D, workspace[3].next=-1, workspace[2].next=3, list.count=4, lastused=3`

delete C: `avail=2, workspace[1].next=3, lastused =3, workspace[2].next =-1`

insert E: `workspace[2].entry=E, workspace[2].next = -1, workspace[3].next=2, avail=-1`

delet A: `list.head=1, list.cout=3, workspace[0].next=-1, avail=0`

delet B: `list.head=2, list.cout=2, workspace[1].next =0, avail= 1`

insert C: `list.cout=3, workspace[1].next=-1, workspace[1].entry=C, workspace[2].next=1, avail= 0`

Prototypes of a linked list (array implementation)

```
void CreateList(List *);
int CurrentPosition(Position, List*);
void DisposeNode(ListIndex, List *);
void Error(char *);
void InsertList(Position,ListEntry *,List *);
ListIndex NewNode(void);
void SetPosition(Position,ListIndex *,List *);
void TraversellList(List *,void(*f)());
void WriteEntry(ListTntry);
```

`ListIndex NewNode(void) //make a new node.`

Pre: The list indices `avail` and `lastused` have been initialized when they were defined as global variables and have been used of modified only by `NewNode` and `DisposeNode`. The `workspace` array is not full.

Post: The list index `newindex` has been set to the first available place in `workspace`; `avail, lastused` and `workspace` have been updated as necessary.

Uses: `avail, lastuse` and `workspace` as global variables.

```
void DisposeNode(ListIndex oldindex,List *list)/* return
a node to available space.*/
```

Pre: The list indices `avail` and `lastused` have been initialized when they were defined as global variables and have been used or modified only by `NewNode` and `DisposeNode`; `oldindex` is an occupied position in `workspace`.

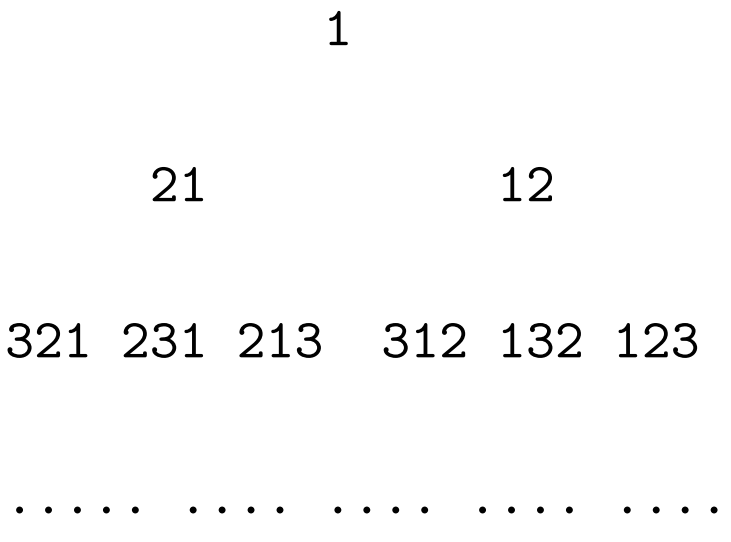
Post: The list index `oldindex` has been pushed onto the linked stack of available space; `avail, lastused` and `workspace` have been updated as necessary.

Uses: `avail, lastuse` and `workspace` as global variables.

```
void TranseList(List *list, void(*f)())
{
 ListIndex current;
 for(current=list->head;current!=-1;
            current=workspace[current].next)
   f(workspace[current].entry);
}
```

```
void InsertList(Position p, ListEntry x, List *list)
{ ListIndex newindex, previous;
  if(p<0 ||P>list->count)
   Error("Inserting into a nonexisting position.");
 else{
  newindex=NewNode();
  workspace[newindex].entry=x;
  if(p==0){
    workspace[newindex].entry=list->head;
    list->head=newindex;
  }else{
    SetPosition(p-1,&previous,list);
    workspace[newindex].next=workspace[previous].next;
    workspace[previous].next=newindex;
  }
  list->count++; }  }
```

Generating Permutations



Idea: to get $n!$ permutations recursively. Suppose we already have all permutaions of size $k-1$. We want all permutations of size k.

```
void Permute(k,n)
{
  for each of the k possible positions in the list
  { Insert k into the given position;
    if k == n
      Process Permutation
    else
      Permute(k+1,n);
    Remove k from the given position;
   }
 }
```

Testing Insert and Delete Logic

1. Insert a node into a null list
2. Insert a node before the first data node
3. Insert between two data nodes
4. Insert after the last node

Test delete logic is similar to testing the insert logic.

Complex Implementations

Circularly linked lists

The last node's link points to the first node of the list.

Keeping the current position: keep a `Position currentpos` in the definition of a `List`. This kind of lists are efficient with applications that refer to the same entry several times or access to nodes in the middle of the list without starting at the beginning.

Insert a node before the first node is the same as insert a node after the last node.

Search a node will be stopped at the node before the start node.

Doubly linked lists

Allows moving both forward and backward.

```
typedef struct listnode{
  ListEntry entry;
  struct listnode *next;
  struct listnode *previous;
} ListNode;

typedef struct list{
  int count;
  ListNode *current;
  Position currentpos;
} List;
```

```
void SetPosition(Position p,List *list)
{
  if(p<0||p >= list->count)
    Error("illegal position.");
  else if(list->currentpos <p)
   for(;list->currentpos!=p;list->currentpos++)
     list->current = list->current->next;
  else if(list->currentpos > p)
   for(;list->current = list->current->previous;
}
```

Doubly linked list uses some space for an extra pointer, but saves search time.

```
void InsertList(Position p, ListEntry x,List *list)
{
  ListNode *newnode,*following;
  if(p<0||p>list->count)
    Error("Attempt to insert in a wrong position.");
  else{
    newnode=MakeListNode(x);
    if(p==0){
      newnode->previous=NULL;
      if(list->count==0)
        newnode->next=NULL;
      else{
        SetPosition(0,list);
        newnode->next=list->current;
        list->current->previous=newnode;
      }
```

```
    }else{
      SetPostion(p-1,list);
      following=list->current->next;
      newnode->next=following;
      newnode->previous=list->current;
      list->current->next=newnode;
      if(following)
         following->previous=newnode;
      }
      list->current=newnode;
      list->currentpos=p;
      list->count++;
    }
}
```

Multilinked lists

A multilinked list is a list with two or more logical key sequences.

A node has two or more link fields according to different key sequences.

To insert a node, need to find out the linking position and insert for each link fields.

To delete a node, need to reconnect the pointers for each logical list.

Example

```
typedef struct node
  {
    int idNumber;
    int telephone;
    string firstName;
    string lastName;
    struct node* nextID;
    struct node* nextName;
   } NODE;
```

```
typedef struct
  {
   int count;
   NODE* IDpos;
   NODE* NamePos;
   NODE* IDhead;
   NODE* IDrear;
   NODE* NameHead;
   NODE* NameRear;
   int (*compare) (void* argu1, void* argu2);
   } LIST;
```