7. Duplán láncolt lista

7.1 Duplán vagy kétszeresen láncolt lista definiálása

Definíció 7.1.1 A duplán vagy kétszeresen láncolt lista olyan adatszerkezet, amelyben minden elem két mutatót tartalmaz: az egyik a következ o elemre mutat, a másik pedig az el oz o elemre mutat.

Előnyei:

1. **Előre és hátra navigálás:**: Mivel minden elemre két mutató mutat, lehetővé teszi az előre és hátra navigálást. Ezáltal könnyedén lehet közlekedni mindkét irányban az elemek között, ami néhány alkalmazási területen hatékonyabbá teheti az adatok manipulálását.
2. **Beszúrás és törlés hatékonysága:**: A duplán láncolt lista lehetővé teszi az elemek közé történő beszúrást és törlést költséghatékonyan. Amikor egy elemet kell beszúrni vagy törölni, a szomszédos elemek mutatóit frissítheted, és nincs szükség az egész listán való végiglépésre.
3. **Hatékony visszavonás:** A duplán láncolt lista esetén az utolsó elem mindig hozzáférhető az utolsó mutató segítségével. Ez a gyors visszavonást teszi lehetővé, mivel a lista végéről való eltávolítás nem igényel teljes lista átvizsgálást.

Hátrányai:

1. **Több memóriaigény:**: A duplán láncolt listák több memóriát igényelnek minden egyes elem tárolásához, mivel két mutatóval rendelkeznek minden egyes elemre (egy mutató az előző és egy a következő elemre). Ez a többletmemóriaigény hátrányos lehet, ha a memóriafogyasztás kritikus szempont.
2. **Bonyolultabb implementáció:**: A duplán láncolt listák implementációja összetettebb, mint a egyszeresen láncolt listáké vagy más egyszerűbb adatszerkezeteké. A kétirányú kapcsolatok kezelése és karbantartása némi bonyolultságot hozhat a kódolás folyamatába.
3. **Nem minden területen hatékonyak:** Bizonyos alkalmazásokban, például az egyszerű keresési vagy iterációs feladatokban, a duplán láncolt listák használata több memóriát és időt vehet igénybe, mint más egyszerűbb adatszerkezetek.

7.2 Alapvető műveletek

1. Inicializálás
2. Bejárás
3. Új elem beszúrása a lista elejére
4. Új elem beszúrása a lista végére
5. Új elem beszúrása megadott helyre (pl. n-ik elem után)
6. Első elem törlése
7. Utolsó elem törlése
8. Megadott tartalmú elem törlése
9. Összes elem törlése
10. Listában keresés
11. Listában elemeinek kiíratása elsőtől utolsóig
12. Listában elemeinek kiíratása utolsótól elsőig
13. Listában keresés

7.3 Duplán láncolt lista. Függvények.

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

7.3.1 Helyfoglalás, kezdőértékadás

```
Node *newNode(int new_data) {
    Node *newItem = (Node*) malloc(sizeof (Node));
    if(!newItem)
    {
        printf("MEMORY_ALLOCATION_ERROR_MESSAGE");
        exit(-1);
    }
    newItem->data = new_data;
    newItem->next = NULL;
    newItem->prev=NULL;
    return newItem;
}
```

A newNode függvény létrehoz egy új csomópontot (Node). A csomópontnak van egy data mezője, amely tárolja az adatot, egy next mezője, amely egy mutató a következő csomópontra és egy prev mezője, ami mutat az előző csomópontra.

A függvény először dinamikusan lefoglal területet a Node struktúrának a malloc függvény segítségével. Ha a memóriafoglalás nem sikerül (a visszaadott mutató NULL), akkor kiír egy hibaüzenetet és kilép a programból.

Ha a memóriafoglalás sikeres volt, akkor az új csomópont data megkapja az új értéket, és a next és prev mezők NULL címet kapnak.

Végül a függvény visszaadja az új csomópontra mutató mutatót.

7.3.2 Üres lista ellenőrzése

```
bool isEmpty(Node *head) {
return head == NULL;
}
```

Az isEmpty függvényellenőrzi, hogy egy láncolt lista üres-e vagy sem. A függvény egy Node típusú fejelem mutatót kap paraméterként.

A függvény összehasonlítja a head mutatót a NULL értékkel. Ha a head mutató NULL, az azt jelenti, hogy a lista üres, és a függvény true értéket ad vissza. Ha a head mutató nem NULL, akkor a lista nem üres, és a függvény false értéket ad vissza.

7.3.3 Új csomópont beszúrása a lista elejére

```
void insertAtBeginning(Node **head_ref, int new_data) {
    Node* temp = newNode(new_data);
    if(isEmpty(*head_ref))
    {
        *head_ref = temp;
        return;
    }
    temp->next = *head_ref;
    (*head_ref)->prev=temp;
    *head_ref = temp;
}
```

Az insertAtBeginning függvény egy új csomópontot helyez a láncolt lista elejére. A függvény két paramétert vár: egy Node** típusú fejelem mutatóra (head_ref), amely a fejelemre mutató mutatót jelöli, valamint egy int típusú new_data paramétert, amely az új csomópontban tárolandó adatot jelenti.

A függvény először létrehoz egy új csomópontot a newNode függvény segítségével, és az new_data értékét adja át neki. Az új csomópontnak a next mezője NULL lesz, mivel azt az új csomópontot az elejére szeretnénk helyezni.

Ezután beállítja az új csomópont next mutatóját a jelenlegi fejelemre mutató mutatóra, hogy a lista elejére helyezze az új csomópontot.

Végül az head_ref mutatóját átállítja az új csomópontra, így az új csomópont lesz a lista új fejeleme.

Fontos, hogy a head_ref mutatót Node** típusként kell átadni, mivel a fejelem mutatójának értékét módosítani kell a függvényben. A dupla mutató használata miatt a fejelem mutatójának módosítása látható lesz minden alprogramban és a főprogramban is.

7.3.4 Új csomópont beszúrása adott csomópont címe után

```
void insertAfter(Node *prev_ref, int new_data) {
    if(prev_ref == NULL)
    {
        printf("NULL_POINTER_EXCEPTION_ERROR_MESSAGE");
        return;
```

```
    }
    Node* new_node = newNode(new_data);

    if (prev_ref->next != NULL) {
        new_node->next = prev_ref->next;
        prev_ref->next->prev = new_node;
    }

    prev_ref->next = new_node;
    new_node->prev = prev_ref;
}
```

Az insertAfter függvény egy új csomópontot szúr be egy meghatározott csomópont után a láncolt listában. A függvény két paramétert vár: egy Node* típusú mutatót (prev_node), amely a meghatározott csomópontra mutat, és egy int típusú new_data paramétert, amely az új csomópontban tárolandó adatot jelenti.

Először a függvény ellenőrzi, hogy a prev_node mutató NULL-e. Ha igen, akkor kiír egy hibaüzenetet (NULL_POINTER_EXCEPTION_ERROR_MESSAGE) és visszatér a függvényből.

Ezután létrehoz egy új csomópontot a newNode függvény segítségével, és az new_data értékét adja át neki.

Az új csomópont next mutatóját beállítja a prev_node->next-re, hogy az új csomópont mutasson az eredeti után következő csomópontra.

Végül a prev_node->next mutatóját átállítja az új csomópontra, hogy az eredeti csomópont után következzen az új csomópont.

Ezáltal az új csomópont beillesztésre kerül a láncolt lista megfelelő helyére a prev_node csomópont után.

7.3.5 Új csomópont beszúrása a lista végére

```
void insertAtEnd(Node **head_ref, int new_data) {
    if(isEmpty(*head_ref))
    {
        *head_ref = newNode(new_data);
        return;
    }
    Node *temp = *head_ref,*last;
    while(temp->next != NULL)
    {
        temp = temp->next;
    }
    last=newNode(new_data);
    temp->next = last;
    last->prev=temp;
    }
```

Az insertAtEnd függvény egy új csomópontot helyez a láncolt lista végére. A függvény két paramétert vár: egy Node** típusú fejelem mutatóra (head_ref), amely a fejelemre mutató mutatót jelöli, valamint egy int típusú new_data paramétert, amely az új csomópontban tárolandó adatot jelenti.

Először létrehoz egy új csomópontot a newNode függvény segítségével, és az new_data értékét

adja át neki.

Ezután létrehoz egy last mutatót, amely az eredeti fejelemre mutat. Ez segít a láncolt lista végét megkeresni.

Ezután ellenőrzi, hogy a lista üres-e a isEmpty függvény segítségével. Ha igen, azaz a head_ref mutató NULL, akkor az új csomópontot állítja be az head_ref mutatóra, és visszatér a függvényből.

Ha a lista nem üres, akkor egy ciklussal továbbítja a last mutatót a láncolt lista végéig (amíg az aktuális csomópont next mutatója nem NULL lesz).

Végül beállítja az utolsó csomópont next mutatóját az új csomópontra, így az új csomópont lesz a láncolt lista utolsó eleme.

Ezáltal az új csomópont beillesztésre kerül a láncolt lista végére, és az head_ref mutató értéke változik a lista elején.

7.3.6 Első csomópont törlése

```
void deleteFromBeginning(Node **head_ref) {
if(isEmpty(*head_ref)) {
printf("NULL_POINTER_EXCEPTION_ERROR_MESSAGE");
return;
}
Node *temp = *head_ref;
*head_ref = (*head_ref)->next;
free(temp);
}
```

A deleteFromBeginning függvény törli a láncolt lista első csomópontját. A függvény egy Node** típusú fejelem mutatót vár paraméterként (head_ref), amely a fejelemre mutató mutatót jelöli.

Először ellenőrzi, hogy a lista üres-e a isEmpty függvény segítségével. Ha igen, azaz a head_ref mutató NULL, akkor kiír egy hibaüzenetet és visszatér a függvényből.

Ha a lista nem üres, akkor létrehoz egy temp mutatót, amely a törlendő csomópontra mutat.

Az head_ref mutatót átállítja a következő csomópontra ((*head_ref)->next), így az új fejelem az eredeti második csomópont lesz.

Végül felszabadítja a temp mutatóval jelölt csomópontot a free függvény segítségével, így törölve azt a láncolt listából.

Ezáltal a függvény törli a láncolt lista első csomópontját, és az head_ref mutató értéke változik a lista új fejelemére.

7.3.7 Utolsó csomópont törlése

```
void deleteFromEnd(Node **head_ref) {
if(isEmpty(*head_ref)) {
printf("NULL_POINTER_EXCEPTION_ERROR_MESSAGE");
return;
}
Node* last = *head_ref, *prev;
while (last->next != NULL)
{
prev = last;
last = last->next;
```

```
}
prev->next = NULL;
free(last);
}
```

A deleteFromEnd függvény törli a láncolt lista utolsó csomópontját. A függvény egy Node** típusú fejelem mutatót vár paraméterként (head_ref), amely a fejelemre mutató mutatót jelöli.

Először ellenőrzi, hogy a lista üres-e a isEmpty függvény segítségével. Ha igen, azaz a head_ref mutató NULL, akkor kiír egy hibaüzenetet és visszatér a függvényből.

Ha a lista nem üres, akkor létrehoz két mutatót: last és prev. Az last mutató az eredeti fejelemre mutat, míg a prev mutatót inicializáljuk, hogy az első csomópont előttre állítsuk.

Egy ciklussal továbbítjuk a last mutatót a láncolt lista végéig, amíg az aktuális csomópont next mutatója nem lesz NULL. A ciklusban az last mutató mindig az aktuális csomópontra, a prev pedig mindig az előző csomópontra mutat.

Miután elérte a lista végét, a prev->next mutatóját beállítjuk NULL-ra, hogy az utolsó csomópontot leválasszuk a láncolt listáról.

Végül a last mutatóval jelölt csomópontot felszabadítjuk a free függvény segítségével, így törölve azt a láncolt listából.

Ezáltal a függvény törli a láncolt lista utolsó csomópontját, és az head_ref mutató értéke változik a lista módosult fejelemére.

7.3.8 Adott információjú csomópont törlése

```
void deleteNode(Node **head_ref, int key) {
    if (isEmpty(*head_ref)) {
        printf("NULL_POINTER_EXCEPTION_ERROR_MESSAGE");
        return;
    }

    if ((*head_ref)->data == key) {
        deleteFromBeginning(head_ref);
        return;
    }

    Node *temp = (*head_ref)->next;
    Node *prev = *head_ref;

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp != NULL) {
        prev->next = temp->next;

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

```
        free(temp);
    }
}
```

A deleteNode függvény törli a láncolt lista egy adott értékkel rendelkező csomópontját. A függvény két paramétert vár: egy Node** típusú fejelem mutatót (head_ref), amely a fejelemre mutató mutatót jelöli, valamint egy int típusú key paramétert, amely az adott értékkel rendelkező csomópontot jelöli.

Először létrehoz két mutatót: temp és prev. Az temp mutató az eredeti fejelemre mutat, míg a prev mutatót inicializáljuk nullpointerrel, hogy az előző csomópontra mutasson.

Ellenőrizzük, hogy az temp mutató nem NULL-e és hogy az temp mutató által mutatott csomópont data mezője megegyezik-e a key értékkel. Ha igen, azaz az első csomópont tartalmazza a keresett értéket, akkor az head_ref mutatót átállítjuk az temp->next-re, azaz az első csomópontot leválasztjuk, majd felszabadítjuk az temp mutatóval jelölt csomópontot a free függvény segítségével. Ebben az esetben visszatérünk a függvényből.

Ha az első csomópont nem tartalmazza a keresett értéket, akkor egy ciklussal keresünk tovább a láncolt listán. Amíg az temp mutató nem NULL és az temp mutató által mutatott csomópont data mezője nem egyezik meg a key értékkel, folytatjuk a keresést. A ciklusban az temp mutató mindig az aktuális csomópontra, a prev pedig mindig az előző csomópontra mutat.

Ha az temp mutató NULL, azaz végigértünk a listán, és nem találtunk csomópontot a keresett értékkel, akkor visszatérünk a függvényből.

Ha megtaláltuk a keresett értékkel rendelkező csomópontot, akkor az prev->next mutatóját átállítjuk az temp->next-re, hogy az előző csomópontot összekapcsoljuk az aktuális csomópont utáni csomóponttal.

Végül felszabadítjuk az temp mutatóval jelölt csomópontot a free függvény segítségével, így törölve azt a láncolt listából.

7.3.9 Adott információjú csomópont keresése

```
bool searchNode(Node* head_ref, int key) {
Node* current = head_ref;
while (current != NULL) {
if (current->data == key) return
true;
current = current->next;
}
return false;
}
```

A searchNode függvény megkeresi, hogy egy adott érték megtalálható-e a láncolt lista csomópontjaiban. A függvény két paramétert vár: egy Node* típusú fejelem mutatót (head_ref), amely a fejelemre mutató mutatót jelöli, valamint egy int típusú key paramétert, amelyet keresni szeretnénk a láncolt listában.

Először létrehozunk egy current mutatót, amelyet az head_ref-re állítunk, azaz az eredeti fejelemre.

Egy ciklussal végigmegyünk a láncolt listán, amíg az current mutató nem lesz NULL. Minden iterációban ellenőrizzük, hogy az aktuális csomópont data mezője megegyezik-e a keresett key értékkel. Ha igen, akkor visszatérünk a függvényből, és igaz értéket adunk vissza, mert megtaláltuk

a keresett értéket.

Ha nem találjuk meg a keresett értéket az aktuális csomópontban, léptetjük az current mutatót a következő csomópontra a current->next segítségével.

Ha végigértünk a láncolt listán, és nem találtuk meg a keresett értéket egyetlen csomópontban sem, akkor visszatérünk a függvényből, és hamis értéket adunk vissza.

Ezáltal a függvény végigmegy a láncolt listán, és megkeresi, hogy az adott érték megtalálható-e valamelyik csomópontban. A visszatérési érték true, ha megtalálja, és false, ha nem találja meg.

7.3.10 Lista kiírása (elsőtől az utolsóig)

```
void printListFromBegin(Node *node) {
    Node* temp = node;
    while(temp != NULL)
    {
        printf("%i -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

A printListFromBegin függvény kiírja a láncolt lista csomópontjainak tartalmát. A függvény egy Node* típusú csomópont mutatót vár paraméterként (node), amely az első csomópontra mutató mutatót jelöli.

Egy ciklussal végigmegyünk a láncolt listán, amíg az node mutató nem lesz NULL. Minden iterációban kiírjuk az aktuális csomópont data mezőjét a printf függvény segítségével.

Ezáltal a függvény végigiterál a láncolt listán, és kiírja a csomópontokban található adatokat.

7.3.11 Lista kiírása (utolsótól az elsőig)

```
void printListFromEnd(Node *node) {
    Node* temp = node;
    while(temp->next != NULL)
    {temp=temp->next;}
    while(temp != NULL)
    {
        printf("%i -> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}
```

A printListFromEnd függvény kiírja a láncolt lista csomópontjainak tartalmát fordított sorrendben. A függvény egy Node* típusú csomópont mutatót vár paraméterként (node), amely az első csomópontra mutató mutatót jelöli.

Egy ciklussal végigmegyünk fordított sorrendben a láncolt listán, amíg az node mutató nem lesz NULL. Minden iterációban kiírjuk az aktuális csomópont data mezőjét a printf függvény segítségével.

Ezáltal a függvény végigiterál a láncolt listán, és kiírja a csomópontokban található adatokat.

7.3.12 Példa a függvények meghívására

```
int main() {
    Node *nNode, *p;
    int x;
    scanf("%i", &x);
    nNode = newNode(x % 10);
    x /= 10;
    while (x != 0) {
        insertAtEnd(&nNode, x % 10);
        x /= 10;
    }
    p = nNode->next;
    insertAfter(p, 100);
    printListFromBegin(nNode);
    printListFromEnd(nNode);
    deleteFromBeginning(&nNode);
    deleteFromEnd(&nNode);
    deleteNode(&nNode, 2);
    printListFromBegin(nNode);
    printListFromEnd(nNode);
}
```

A fenti main függvény egy egyszerű példa arra, hogyan hozzunk létre egy láncolt duplán láncolt listát egy szám számjegyeiből, majd elvégzi a következő műveleteket:

insertAfter(p, 100);: Beszúr egy új csomópontot a p csomópont után, melyet felruház a 100-as értékkel.

printListFromBegin(nNode); printListFromEnd(nNode): Kiírja a láncolt listát mind az elejéről, mind pedig a végéről.

deleteFromBeginning(nNode); deleteFromEnd(nNode);: Törli az első és az utolsó csomópontot láncolt listából.

deleteNode(nNode, 2);: Törli az összes 2-es értékű csomópontot a láncolt listából.

Ezen műveletek együttesen tesztelik a láncolt lista műveleteit, például a beszúrást, törlést, kiírást.

7.4 Megoldott feladat

1. Írj egy programot, amely eldönti egy megadott számról, hogy tükörszám-e, duplán láncolt lista segítségével.

Megoldás:

```
 int isSymmetric(Node* head) {
   Node* tail = NULL;
   Node* current = head;
          while (current!=NULL && current->next!=NULL) {
       current = current->next;
   }

   tail = current;

   while (head!=NULL && tail!=NULL) {
```

```
        if (head->data != tail->data) {
            return 0;
        }
        head = head->next;
        tail = tail->prev;
    }

    return 1;
}
 int main()
 {
 Node *nNode, *p;
    int x;
    scanf("%i", &x);
    nNode = newNode(x % 10);
    x /= 10;
    while (x != 0) {
        insertAtEnd(&nNode, x % 10);
        x /= 10;
    }

  if(isSymmetric(nNode))
    printf("A lista szimmetrikus");
  else
      printf("A lista nem szimmetrikus");
}
```

Megjegyzés: A változók deklarálása, valamint az isSymmetric függvényen kívül használt többi függvény definíciója a fent leírtakkal azonos.

7.5 Javasolt kérdések

1. Mit ír ki az f1 függvény, ha p a lista első elemének a címe és a lista elemei: 6 2 2 4 5 6 6 6 7 és a lista csomópontjának struktúrája:

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

void f1(Node* head) {
Node* tail = NULL;
Node* current = head;
     while (current!=NULL && current->next!=NULL) {
        current = current->next;
    }
    tail = current;

    if (head!=NULL && tail!=NULL) {
        head = head->next;
        tail = tail->prev;
```

```
    }
printf("%i %i",head->data,tail->data);
}
```

2. Az f2 függvény megkapja paraméterként egy duplán láncolt lista első elemének a címét és ki kell írnia a lista elemeit, elindulva a végétől az elejéig, viszont hibákat tartalmaz. Keresd meg a hibákat és javítsd ki!

```
typedef struct Node{
    int data;
    struct Node* next, *prev;
}Node;
...
void f2(Node *node) {
    Node* temp = node->next;
    while(temp->next != NULL)
    {temp=temp->next;}
    while(temp != NULL)
    {
        printf("%i -> ", temp->next);
        temp = temp->next;
    }
}
```