Programmering 3 – Lab 1:

Implementasjon av dobbeltlenket sirkulær liste i C++

22.09.2024

Øyvind A. Nordbø

**Introduksjon.**

**Oppgaven**:

Implementere en dobbelt lenket sirkulær liste som støtter følgende operasjoner:

1. Legge til et element i listen.
2. Slette et element fra listen ved bruk av enten ID eller navn.
3. Skrive ut alle elementene i listen fra start til slutt, i sortert rekkefølge basert på ID.

*Kommentar*:

I punkt 3 over, ba oppgaven mer spesifikt om å lage funksjonen «skrivUt()». Som det vil vises, hadde jeg før jeg leste den delen av oppgaven, allerede lagd en overlastet funksjon for utskrifts-operatoren på «DobbeltLenketSirkuaerListe» objekter. Oppgaven hadde vel tenkt at den skulle implementeres som en klasse funksjon, men kravene er tilfredsstilt, og syntes ikke det var nødvendig å lage en ny funksjon.

**Data strukturen.**

For å lage en dobbelt lenket sirkulær liste, brukte jeg 2 klasser:

**1.Class Node.**

Brukes for å lage nye objekter i listen. Alle objekter skal inneholde 2 shared pointers («NEXT» og «PREV») som hver peker til forrige og neste element i listens rekkefølge. Slik sikrer jeg at listen er dobbelt lenket, at vi enkelt vha. Pekerne kan bevege oss fram og tilbake i listen.

**2. Class DobbeltLenketSirkukaerListe.**

Inneholder pekerne («HEAD» og «TAIL») til første og siste element i lista. Dette gjør det enklere å få tak i første og siste element, og beholde strukturen etter at listen har blitt opprettet. Koblingen mellom de to ende punktene, gjør at vi kan bevege oss fra slutt til start, og motsatt.

Dermed kan vi opprettholde kravene, og sørge for at listen kan være på sirkulær dobbelt lenket listeform.

Konstruktører:

Når listen først blir opprettet, er den selvfølgelig tom, og vi setter pekerne til nullptr. Da HEAD og TAIL ikke har noen objekter å peke på enda.

**Funksjons implementasjoner.**

**LeggTil.**

For funksjonen leggTil(), opprettes først et nytt objekt ved hjelp av std::make\_shared<Node>(id, name).   
Hvis HEAD er null, som betyr at listen er tom, settes både HEAD og TAIL til dette nye objektet. Her settes både PREV og NEXT til HEAD, slik at objektet peker til seg selv og opprettholder den sirkulære egenskapen.   
Hvis listen ikke er tom, settes PREV til TAIL (det nåværende siste objektet) og NEXT til HEAD (det første objektet). Deretter oppdateres HEAD->PREV til å peke på det nye objektet, og TAIL->NEXT settes også til det nye objektet. Til slutt oppdateres TAIL til å bli det nye objektet.

**Slett.**

Når det gjelder funksjonen slett(int id), begynner den med å sjekke om listen er tom. Hvis HEAD er null, returnerer funksjonen tidlig. En løkke itererer deretter gjennom listen for å finne objektet med den spesifiserte id.   
Hvis objektet finnes, oppdateres pekerne for å fjerne det fra listen. Hvis objektet som skal slettes er HEAD, oppdateres HEAD til temp->NEXT.   
Hvis den nye HEAD er det eneste objektet i listen, settes både HEAD og TAIL til null.   
Hvis objektet er TAIL, oppdateres TAIL til temp->PREV, og TAIL->NEXT justeres til å peke på HEAD. Hvis objektet er en midt-objekt, oppdateres temp->PREV->NEXT og temp->NEXT->PREV for å hoppe over det slettede objektet.

**Merge Sort.**

Merge sort brukes til å holde listen sortert på ID etter at nye elementer er lagt til eller fjernet. Dette er en effektiv algoritme for sortering som har en tidskompleksitet på

Big O(nlog n). Funksjonene som implementerer merge sort i denne klassen er delt inn i flere deler:

1. lagVektor()

For å enklere håndtere alle objektene under merge sort, valgte jeg å putte de inn i en vektor. Do while løkken lærte jeg meg fort nytten av igjen. En temp peker av hodet blir laget. Deretter har vi en do while betingelse som er gyldig når temp brukeren ikke er lik hodet. Pekeren blir lagt til i vektoren, og deretter settes den til å være det objektet NEXT pekte på. Dermed er while betingelsen oppfylt og dette gjøres til den har gått fra å være TAIL til HEAD igjen. Dermed har vi en vektor vi kan bruke for å dele opp i nye vektorer, og gjennomføre merge sort.

2. mergeSort()

Denne funksjonen starter merge sort-prosessen. Den tar et alias av den usorterte listen, og i funksjonen blir 2 nye lister laget av denne: «left» og «right». Vi finner midten, «halfway» av listen hver gang ved å lagre «list.size() / 2» i en variabel. Deretter, fyller vi den ene listen fra «list.begin()» til «list.begin() + halfway». Den andre fra «list.begin() + halfway» til «list.end()».   
Her var tanken at vi kaller en ny funksjon som faktisk sammenlikner de 2 splittede listene, sorterer de på ID, og legger de sammen som en del sortert liste. I «mergeSort()», returnerer vi resultatet av nettopp dette og kaller funksjonen «merge()» her. Det som imidlertid skjer før her, er at «mergeSort()» blir kalt på de to splittede listene, «left» og «right» igjen. Dette vil altså gjentas (rekursivt funksjonskall) helt til if-setningen, som; sjekker om listen sin «.size()» er mindre enn eller lik 0, og returnerer listen før funksjonen kommer til linjen hvor den kaller seg selv igjen på de to listene.

3. merge()

Funksjonen tar 2 vektorer som parameter, og vil som forklart over, få «left» og «right» vektorer fra mergeSort() sin return. En ny vektor («sortedVec») hvor vi skal lagre den kombinerte og sorterte vektoren blir opprettet. For sortering og sammenslåing, opprettes først to indekser i og j til å gå gjennom left og right. Disse får begge verdien 0 i begynnelsen. I while sammenliknes ID’en til objektene i de to listene.

* Hvis ID-en til noden i left er mindre enn ID-en til noden i right, legges noden fra left til sortedVec, og vi går videre til neste node i left (i++).
* Hvis ikke, legges noden fra right til sortedVec, og vi går videre til neste node i right (j++).

Dette gjøres på betingelsen at «i» og «j» begge er mindre enn «.size()» av hver sin vektor. Det som dermed kan bli et problem, er at vi har gått gjennom alle elementene i én av delene (for eksempel hvis «i» er lik left.size()), vil betingelsen bli false, og dermed stopper løkken.Da kan det hende at det fortsatt er elementer igjen i den andre del-listen. Derfor løste jeg det enkelt ved å definere 2 nye while løkker, som enkelt sier at så lenge i er mindre enn listen sin størrelse, skal vi legge til «i/j++» i sortedVec for hver gang i loopen, og dermed få med de siste elementene. Når dette er gjort blir listen returnert ferdig sortert.

4. doMergeSort()

doMergeSort() kalles for å starte sorteringsprosessen. Den konverterer først listen våres til en vektor ved å kalle lagVektor() funksjone. Deretter bruker jeg mergeSort() funksjonen på lista, og lagrer denne i en egen variabel. Her valgte jeg igjen å sende listen til en ny funksjon, «oppdaterListe()», som har som oppgave å oppdatere listen våres etter vektoren vi har sortert.

5. oppdaterListe()  
Tar som nevnt et alias av vektor pekere av objektet mitt som parameter, og får den sorterte listen fra doMergeSort(). Her bruker jeg aller først en if, som sjekker om listen er tom og returnerer den hvis den er det for sikkerhet. I de neste linjene setter jeg listen min sin HEAD lik det som liger på indeks plass nummer 0, altså det første objektet i den sorterte listen. TAIL settes lik listen sin «.back()» som vil være det siste objektet i listen.

Deretter har jeg en for-løkke, som går gjennom alle objektene i den sorterte listen én etter én. For hvert objekt skjer følgende: Objektet på indeks plass «i» blir lagret. For å fikse pekerne sin «PREV» og «NEXT», klarte jeg ved å trikse med iteratoren, lengden på listen, og modulo. Indeksen «(i + 1) % sortedNodes.size()» i vektoren, tar den nåværende indeksen og legger til 1 for å finne det neste Objektet. Hvis vi er på den siste noden (i == sortedNodes.size() - 1), vil dette gi oss indeksen til den første noden (0), fordi vi bruker % sortedNodes.size().