Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2022-09-09

Lista

Förkunskaper

Samtliga förkunskaper från Klockslag. Dessutom kan du skriva en väl inkapslad klass med grundläggande konstruktor och operatorer. Du kan dela upp din kod i olika filer och vet hur varje fil kan kompileras. Du vet hur flera kompilerade filer kan sättas samman till ett program.

Scenario

Du jobbar kvar på Saab och efter din lyckade programmodul för ett klockslag har du nu fått uppgiften att ta fram en programmodul för en länkad lista. Det är dina kollegor som ska använda din programmodul för att skriva ett flertal olika program, och de har krav på att insättning först och sist i listan alltid ska ske i konstant tid oavsett hur lång listan är. Vidare ska det inte ställas några krav på programmeraren som använder listan för att användningen ska ske korrekt och utan minnesläckor. Programmodulen för listan ska skrivas i form av en klass.

Kvalitetskrav

Samma kvalitetskrav som i Klockslag gäller. *Dessutom gäller nya krav skrivna i kursiv stil.*

Korrekthet

Dina kollegor ska inte kunna använda listan på fel sätt. Det ska enbart gå att skapa fullständigt korrekta listor. Felaktig användning av din modul ska leda till kompileringsfel i första hand och körfel i andra hand. Din modul ska alltid generera fullständigt korrekt resultat. Din modul ska aldrig generera oväsentligt eller fel resultat. Fel som uppstår när listan används ska rapporteras till anropande kod. Anroparen ansvarar för att filtrera vad som rapporteras till slutanvändaren. Du behöver alltså fundera på extra noga på hur din kollega använder din kod och hur hen ska detektera fel som uppstår. Din modul ska ha testfall som alla verifieras av ett helautomatiskt testprogram. Varje del i din klass ska testas utförligt av ditt testprogram.
Oavsett hur din lista används ska den aldrig läcka minne eller orsaka minnesfel. En kombination av ett fullständigt testprogram och kontroll med Valgrind hjälper dig kontrollera detta. Listan ska tillåta att dubbletter av redan närvarnade värden sätts in. Fel som uppstår vid användning av listan ska rapporteras till anropande kod med hjälp av ett lämpligt undantag.

Användbarhet

Konstanta listor ska gå att använda så länge de inte ändras. Programkoden för listan ska vara samlad i en implementationsfil med tillhörande headerfil. Det ska vara tydligt vilken kod som är avsedd för dina kollegor att använda och vilken kod de inte ska röra. Användning av ej avsedd kod ska leda till kompileringsfel. Din modul ska följa C++ konventioner där det är möjligt. Din modul ska inte ge upphov till kompileringsvarningar eller kompileringsfel.

Ändringsbarhet

Du ska förutsätta att all din kod som är avsedd att användas av dina kollegor också används i flera olika andra program. Du ska se till att du kan ändra den interna representationen av listan och att en sådan genomgripande ändring av din modul inte påverkar dina kollegors program över huvud taget. Deras program ska inte märka någon skillnad i syntax eller funktion mellan din gamla och din nya modul. Dina kollegor behöver alltså inte ändra något trots att du ändrat massor.

Prestanda

Följande krav gäller oavsett längd på listan. Insättning först och sist ska i konstant tid (oberoende längden på listan). Kopiering av hela listan ska ske i linjär tid (direkt proportionell mot längden av listan). När så är möjligt ska kopiering ske i konstant tid (flytt-semantik).

Funktionalitetskrav

Det ska finnas funktioner i din klass för att:

- Skapa en tom lista.
- Sätta in ett nytt värde först i listan.
- Sätta in ett nytt värde sist i listan.
- Ta bort värdet först i listan.
- Ta bort värdet sist i listan.
- Ta reda på om listan är tom.
- Konvertera till en sträng på formen: `[2, 11, 8, 7]`.
- Hämta ut första värdet i listan (`front()`).
- Hämta ut sista värdet i listan (`back()`).
- Hämta ut värdet på angiven position i listan (`get(int)`).
- Tömning av listan (borttagning av alla element).
- Ta reda på listans längd (antal insatta element).
- Kopiera en hel listan via vanlig tilldelning (operator=) och via kopieringskonstruktor.

Utöver ovan lista på funktioner som är tänkta att användas direkt av användaren kommer din lista att behöva ett antal extra funktioner för att lösa korrekthetskrav och prestandakrav, samt ett utförligt testprogram för att bevisa att allt fungerar som det ska.

Uppgift

Du ska visa att du förstått hur C++ koncept används och fungerar genom att implementera programmodulen så att alla typer av krav uppfylls. Du redovisar muntligen för din assistent under labtid. Därefter lämnar du in digitalt för bedömning. Koncept som ingår listas nedan.

Bonusuppgift: Sortering av listan [6p]

Lägg till en funktion för att sortera din länkade lista. Sorteringen ska ske så effektivt som möjligt och in-place (får inte kräva extra minnesutrymme för den lagrade datan). Det kan vara frestande att använda en populär och välkänt effektiv sorteringsalgoritm, men i detta fall måste du tänka på vilken effekt avsaknaden av effektiv random-access har på din implementation. En i generella fallet långsammare algoritm kan vara avsevärt effektivare för en länkad struktur.

Koncept som ingår

Alla listade koncept (utom frivilliga) ingår i uppgiften. Endast listade koncept ingår i uppgiften.

- Pekare (namngiven direkt variabel som pekar ut en indirekt anonym variabel)
  - Innehållsoperatorn, avreferering med *
  - Piloperatorn, medlemsåtkomst med ->
  - Adressoperatorn & (använd inte för annat än adressjämförelser, blanda ej ihop med referensdeklaration &)
- Inre klasser
- Speciella medlemsfunktioner
  - Konstruktorer - defaultinitiering eller från andra värden/typer (constructor)
  - Kopieringskonstruktor - djup kopiering till nytt objekt (copy constructor)
  - Kopieringstilldelning - djup kopiering till befintligt objekt (copy assignment)
  - Flyttkonstruktor - snabb flytt från döende till nytt objekt (move constructor)
  - Flyttilldelning - snabb flytt från döende till befintligt objekt (move assignment)
  - Destruktor - djup destruktion (destructor)
- Algoritmanalys (OpenDSA kapitel 2)
- Linjära datastrukturer (OpenDSA kapitel 4.1-4.4)
- Användning av valgrind för att upptäcka minnesläckor. (Kort guide till Valgrind)
- Kasta och fånga undantag (`throw`, `std::logic_error`, `CHECK_THROWS()` alt. `try{} catch{}`)

Tillvägagång

1. Läs på och lär dig hur respektive koncept kan användas för att uppnå kraven i scenariot. Nedan finns förberedelsefrågor och reflektionsfrågor.
2. Planera vilka koncept som behövs till respektive funktionalitet.
3. Se tipsavsnittet nedan för figurer och lämplig implementationsordning.
4. Använd testdriven utveckling(TDD) och lös en sak i taget. **Kontrollera tipsen.**
5. Utvärdera om du uppnått varje krav. Nedan finns frågor för självbedömning.
6. Redovisa och lämna in.
7. Din assistent utvärderar om du uppnått kraven. Brister ger komplettering.

Förberedelsefrågor

- Vad är skillnaden mellan en referens och en pekare?
- Vad innebär en adress när vi talar om pekare? Hur kommer vi åt den?
- Vad är skillnaden mellan en flytt och en kopiering?
- Vad är skillnaderna mellan public, protected och private?

Reflektionsfrågor

- Varför behöver vi en destruktor?
- Vad händer om vi inte deklarerar en konstruktor eller destruktor?
- Vad innebär det att ett objekt dör?
- Varför behöver vi hantera minne? Vad kan hända med vårt program och dator om vi inte gör det?
- Varför har vi inre klasser?
- Varför är inkapsling ett viktigt koncept?

Frågor för självbedömning

Om svaret på någon av nedan frågor är "ja" ger det med all sannolikhet komplettering.

- Får du fel eller varningar när du kompilerar med alla kursens flaggor?
- Har du inkluderat en cc-fil någonstans?
- Har du skrivit `using namespace` i din h-fil?
- Har du använt nyckelordet `friend`?
- Har du använt `cout` eller `cerr` någonstans i din klass?
- Får du kompileringsfel om du inkluderar din h-fil två gånger?
- Kan du i huvudprogrammet få ett objekt av din klass att vara en ogiltig lista?
- Kan du i huvudprogrammet få ett objekt av din klass att innehålla ett för programmeraren oväntat (men giltigt) värde?
- Är någon publik medlemsfunktion svårbegriplig eller irrelevant för den som ska använda klassen?
- Kan du ge argument med odefinierad betydelse till någon funktion?
- Skiljer sig parametertyp eller returtyp från C++ standard för någon av dina speciella medlemsfunktioner?
- Har du någon in-parameter av klasstyp som kopieras eller kan ändras?
- Har du någon medlemsfunktion som aldrig ska ändra sitt objekt, men kan råka göra det utan att det ger kompileringsfel?
- Finns kod som utför samma sak på flera ställen?

- Finns någon konstruktor som har en inkomlett datamedlemsinitieringslista?
- Finns någon publik datamedlem?
- Kan du koppla varje listat koncept till något du skrivit i koden?
- Är det olika indentering eller kodstil på olika ställen i koden?
- Saknar du testfall med förväntade indata för någon funktion?
- Saknar du testfall med oväntade indata för någon funktion?
- Saknar du testfall som utvärderar gränsfall för någon funktion?
- Finns det någon rad kod som aldrig körs till följd av dina testfall?
- Får du minneläckor när du kör ditt testprogram med valgrind?
- Kan du i huvudprogrammet skapa ett objekt av den inre node-klassen?
- Saknas någon av de fem speciella medlemsfunktionerna?

Implementationsordning och figurer

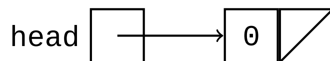
Börja fundera på hur funktionalitetskraven ska användas från ett huvudprogram. Hur skulle du vilja skriva koden som skapar en lista, läser in 10 tal som sätter in först i listan, och sedan skriver ut alla tal i listan genom att hämta ut och ta bort det sista talet tills listan är tom?
Utifrån det du kommer fram till kan du skapa det synliga gränssnittet till hur listan används. Du får fram vad funktionerna ska heta för att huvudprogrammet ska bli tydligt, vilka parametrar de ska ta och vilka returvärden du vill ha. Du kan nu skriva deklarationer för allt synligt i klassen, dvs en stor del av headerfilen.
Nästa steg är att fundera på hur listan ska organiseras i datorns minne. Det måste alltid gå att lägga till ett nytt värde i listan både först och sist utan att tidigare värden behöver flyttas runt (se prestandakrav). Sättet du **SKA** lösa detta är att skapa en länkad struktur av dynamiskt allokerade hjälpobjekt. Dessa hjälpobjekt ska *inte* vara synliga för den som använder listan.

> För att förstå hur listan ska hänga ihop anordnar vi i detta stycke en skattjakt (för åldern ca 5 år). Skatten är en godispåse och du tänker ut ett bra gömställe. Sedan ritar du en "skattkarta" som blir en ledtråd till gömstället. Du inser dock snabbt att detta inte är tillräckligt. När barnet får ledtråden kommer det ta typ 5 sekunder så är skatten hittad.
> Men så kommer du på en genial idé! Du gömmer även ledtråden, tillsammans med en uppmuntrande godisbit. Sedan ritar du en ny ledtråd till den nya gömman. Nu är skattjakten dubbelt så spännande och barnet har att göra åtminstone 10 sekunder. Du suckar inombords när du inser hur många gömmor du måste komma på och hur många ledtrådar du måste rita för att hålla barnet upptaget i åtminstone två minuter, men skrider ändå glatt till verket.
> När du är inne på elfte skattgömman kommer du på att du vill den ska hittas näst sist. Du suckar djupt inför arbetet att flytta om alla gömmor och

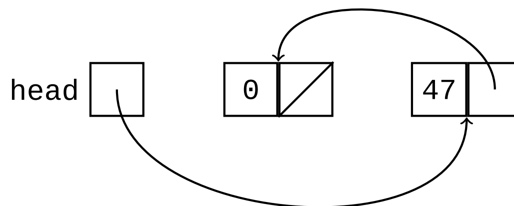
kartor för att lägga in en ny gömma näst sist innan du slås av att det behöver du ju inte alls göra! Vad behöver du egentligen som minst ändra för att lägga in en ny gömma näst sist?
Perfekt tänker du till slut, första ledtråden leder till en godisbit och ny ledtråd, som leder till ny godisbit och ny ledtråd och så vidare tills sista ledtråden leder till resten av godispåsen (och ett "grattis, du har slutfört skattjakten"). Förutom att barnet inte får allt godis på en gång får det springa runt en massa för att hitta allt om du placerar gömmorna omväxlade på var sin sida om trädgården eller huset.

Okej. Nu vet du principen. Vi gör samma sak med den data vi vill lagra i listan. Vi hittar en plats i minnet där vi kan "gömma" (lagra) data så länge vi vill (med `new`). Där lägger vi det värde vi vill lagra och en ledtråd (pekare) till nästa gömma. Allt vi behöver hålla reda på för att hitta hela sekvensen är ledtråden (pekaren) till första gömman. För att förstå precis hur det fungerar **SKA** du rita upp en lista/skattjakt på papper och med papper och penna öva på att lägga till och ta bort värden/godisgömmor. Utan visuellt stöd kommer den mänskliga hjärnan till korta när det gäller att manipulera en länkad lista på rätt sätt. Det finns visserligen de människor som utan att rita upp problemet på papper - till slut - reder ut det, det är bara det att det går så mycket snabbare att lägga tid på att rita upp det, speciellt vid felsökning.
I kursen har vi ett gemensamt sätt hur vi ritar upp listor så att alla lättare kan förstå en figur. Så här ritar vi ut en tom pekarvariabel (slut-på-jakten-ledtråd):

Så här ritar vi listan efter att ett nytt värde (0) satts in (ledtråd till en gömma):

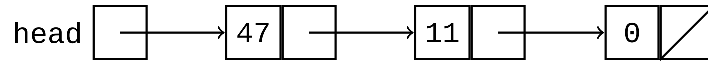


Så här ritar vi listan efter att ytterligare ett värde (47) satts in först:

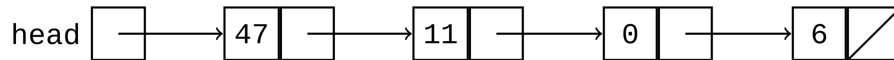


Ofta hyfsar vi till ovan bild för tydlighets skull. Strukturen är fortfarande densamma, men vi ritar noderna i ordning:

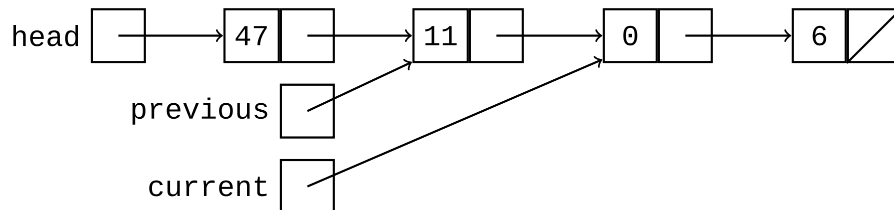
Så här ritar vi listan efter att ett ytterligare ett värde satts in i mitten:



Så här ritar vi listan efter att ett ytterligare ett värde satts in sist:



Tänk på att du och datorn inte har samma perspektiv på listan. Datorn kommer (liksom 5-åringen) bara känna till första pekaren (första ledtråden). Även om DU direkt vet vilken plats i listan du vill ändra så måste datorn först räkna sig fram dit (precis som 5-åringen skulle behöva leta sig fram steg för steg). Så här kan det se ut när programmet sökt sig fram till den treje platsen i listan med två hjälpvariabler (previous och current):



Tips

Labbens kompileringsflaggor

```
-std=c++17 -Wall -Wextra -pedantic -Weffc++ -Wold-style-cast
```

Kompilering med Catch

Testramverket i `catch.hpp` är stort och du märker snart att det tar lång tid att kompilera. Därför rekommenderas det att du förkompilerar enbart testramverket för att nästa kompilering återanvända den redan kompilerade filen. Du gör detta i två steg; gör först ett huvudprogram som inkluderar testramverket och startar det (givet som `test_main.cc`), och länka sedan ihop det med dina testfall du skriver på en separat fil (t.ex. `list_test.cc`). Här är de kommandon som krävs. Du måste själv lägga till de flaggor som behövs i enligt ovan.

1. Kompilera filen `test_main.cc` men länka inte (skapa inte ett körbart program). Använd kompileringsflagga `-c` för att kompilera utan att länka: `g++ -std=c++17 -c test_main.cc` Du kommer nu få en fil `test_main.o` om allt gått bra. Detta är en s.k. objektfil med förkompilerad kod redo att länka.
2. Lägga nu till testfall i en separat fil, `list_test.cc`, och din implementation i filerna `List.h` och `List.cc`. När du vill kompilera ditt testprogram lägger du till den förkompilerade filen på kommandoraden istället för källkodsfilen så inkluderas den i länkningen. Källkodsfilerna kommer kompileras innan de länkas som vanligt:

   ```
   g++ -std=c++17 test_main.o List.cc list_test.cc
   ```

Filuppdelning

Det här är kontroller du kan göra själv. Det är inte del i något testprogram. Följande terminal-kommandon ska aldrig ge några träffar (det ska inte finnas gch-filer, h-filer ska inte tvinga på användaren namnrymder, cc-filer ska aldrig inkluderas):

```
ls -1l *.gch
grep "using namespace" *.h
grep "#include.*ċc" *.cc
```

För att kontrollera din inkluderingsgard kan du helt enkelt prova att inkludera din h-fil två ggr:

```
#include "my_module.h"
#include "my_module.h"
```