OSM: G1

Maria Caroline Miller, 040779, twq135
Søren Pilgård, 190689, vpb984

20. februar 2012

Indhold

1 Opgave 1 - Hægtede lister - list.c og list.h

1.1 Implementation af length and head

Length skal finde længden af en liste, og returnere denne. Med en simpel while-løkke løbes listen igennem, indtil pegeren peger på NULL. For hver gang lægges 1 til variablen len, og pegeren flyttes til næste element i listen med start = start→next.

Head svarer til det første element i en given liste. Dette kan derfor nemt gøres ved returnere start→content, som indeholder første elements data. Det er dog vigtigt lige at tjekke om listen eventuelt er tom, og i så tilfælde returnere NULL.

1.2 Implementation af append og prepend

Prepend tilføjer et element forrest i den hægtede liste. Til at starte med kræves det at man laver plads til et nyt element ved hjælp af malloc. Vi har valgt at kopiere out_of_memory fra den polske lommeregner fra øvelsestimen, og denne afslutter programmet hvis der ikke er mere plads i hukommelsen. Derefter oprettes elementet med sin data og dens next-peger sættes til at pege på det gamle første element, specificeret ved *start. Derefter sættes *start til at pege på den nye element.

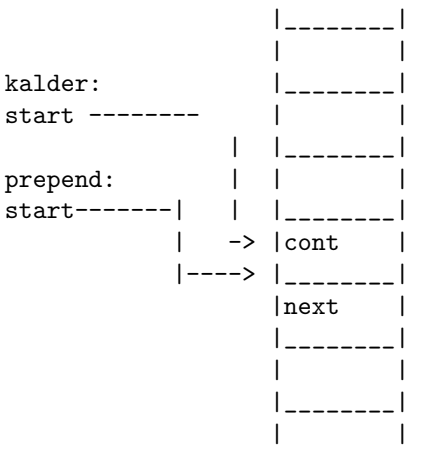
Append tilføjer et element i slutningen af listen. Igen skal der laves plads til et nyt element ved hjælp af malloc, og det nye element oprettes med data og peger. Hvis listen er tom, kalder vi prepend, som tilføjer et element i starten af listen. Ellers skal vi løbe listen igennem indtil vi finder et element hvis next-peger peger på NULL. Dette er det sidste element i den allerede oprettede liste. Dens peger sættes til at pege på det nys oprettede element istedet for NULL.

1.3 Årsager til brug af typen Listnode** start

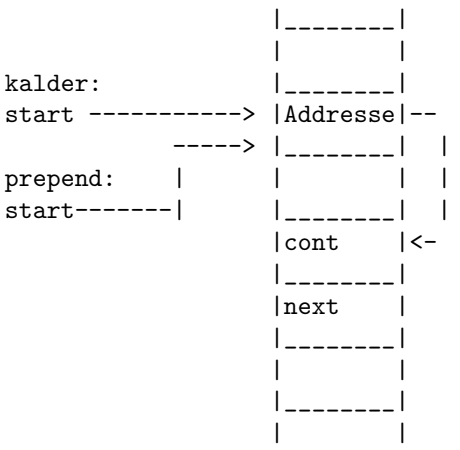
I figur 1 ses hvordan en kaldende funktion kalder prepend med en peger direkte til første element. Prepend virker ved at oprette en ny hægte med den tidligere forreste hægte som dens næste. Problemet er dog at vi fra prepend-funktionen ikke kan gøre den kaldende funktion klar over at listen ikke længere starter det samme sted. Prepend's startvariabel er lokal så ændringer propageres ikke tilbage til kalderen som stadig ser det samme element som det første.

I figur 2 ses løsningen. Her kaldes prepend med en peger til en peger i hukommelsen som peger til det første element i listen. Så selv om ændringer af start ikke propageres tilbage til start kan vi ændre i det som start peger på. Altså addressen til det første element i listen.

I normale tilfælde vil det ikke være nødvendigt med en pegerpeger som ar-



Figur 1: prepend(Listnode *start, Data elem)



Figur 2: prepend(Listnode **start, Data elem)

gument til append. Append skal bare vide hvor listen starter og følge den til den når enden hvor den så indsætter et nyt element. Men fordi append også skal kunne håndtere at indsætte elementer i en tom liste er vi nødt til at bruge en pegerpeger. Ellers ville vi stå i den situation at vi ikke kunne ændre listen til at pege på det første element. Problemstillingen er dermed tilsvarende som for prepend. Dette er især tydeligt da vi i vores løsning har sagt at hvis der kaldes append på den tomme liste svarer det til en prepend, så vi kalder blot prepend-funktionen.

1.4 Remv

I den udleverede kode til remv løbes listen igennem indtil det første match på data mødes, og hvis et sådant findes slettes dette element i listen, og der returneres. Hvis ikke gør funktionen ikke noget. Der er dog et problem med while-løkken i funktionen - eller rettere med rækkefølgen af betingelserne til while-løkken. I den udleverede version tjekkes der først på om det nuværende elements data matcher, og derefter på om det nuværende element er NULL. Hvis man forestiller sig at man har løbet hele listen igennem uden at finde et match, og nu befinder sig på sidste element. Dette passer heller ikke, og while-løkken rykker videre til næste element. curr bliver herefter sat til det som curr→next peger på - i dette tilfælde NULL. Når man næste gang skal kigge på betingelserne i while-løkken prøver den hermed at lave et match på NULL, da curr→content ikke eksisterer i NULL. Den anden betingelse tester på om curr==NULL. Det er derfor ønskværdigt at de to betingelser bytter plads, da C's && operator garanterer venstre-mod-højre evaluering, og vi derfor kan sikre os mod at vi prøver at køre match på NULL.

Vi har derfor rettet remv, så der nu er byttet om på de to betingelser i while-løkken.

1.5 Brug af vores hægtede liste som stack til calc.c

For at optimere implementationen af stakke har vi valgt at bruge en liste forfra. Så istedet for at sætte nye elementer bag på listen, sættes de foran med prepend. Da man i en traditionel stak kun arbejder med det øverste element er der ingen grund til at traversere hele listen for at pushe og poppe. Ved altid at indsætte nye elementer forrest betyder det at push, pop og top kan gøres i konstant tid i modsat til lineærtid hvis hele listen skulle traverseres. For nemhedens skyld har vi dog implenteret empty med length funktionen som betyder at den kører i lineær tid selv om det kunne gøres med en test om der eksisterer bare et element i listen.

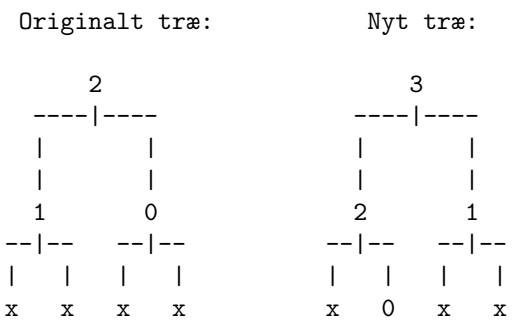
stack_init er en dummy funktion der intet foretager sig. I modsætning til den oprindelige implementation er der ikke noget data der skal initialiseres. Men for at overholde headerfilen har vi implementeret en funktion der ikke gør noget.

stack_empty er en simpel test om lengden af listen er lig med 0.

stack_top er et kald til head.

stack_push er et kald til prepend. Da liste implementationen har sin egen fejl håndtering hvor programmet afsluttes hvis der ikke kan allokeres plads vil prepend funktionen have allokeret plads korrekt hvis den returnerer og vi kan derfor altid returnere 0.

stack_pop er lidt mere krævende da der ikke findes en tilsvarende funktion i list.h. funktionen river derfor manuelt hovedet af listen, deallokerer hukkomel-



Figur 3: addL: nyt element indsat

sen, sætter stackpegepinden til at pege på det nye øverste element og returnerer indholdet af det tidligere hoved.

2 Opgave 2 - Fleksible arrays med Brauntræer

Brauntræer er binære træer, hvor hvert undertræ har det samme antal knuder, eller det venstre har kun en mere end det højre. Disse træer skal bruges til at implementere fleksible arrays.

2.1 Implementation af addL

Funktionen addL skal indsætte et nyt element i starten af array'et. Dette vil sige at elementet skal indsættes som den nye rod i træet, og derefter skal resten af træet ordnes, så det stadig er et Brauntræ. Til at starte med skal der laves plads til det nye element med malloc, og elementet skal oprettes med data og pegere. Hvis træet er tomt er det nemt, da elmentet bare skal indsættes som det første element. Når man indsætter elementer i et Brauntræ skal det sikres at træet hele vejen ned igennem overholder reglerne for dette, og derfor er man nødt til at flytte rundt på de forskellige grene af træet, når der indsættes et nyt element. Det nye element/den nye rod skal have den gamle rods venstre undertræ som sit højre undertræ. Derudover skal den gamle rod og dens højre undertræ lægges ind som den nye rods højre undertræ. Dette giver et noget skævt balanceret træ, som det kan ses nedenfor.

Det er derfor nødvendigt rekursivt at gennemløbe træets venstre side for at sørge for at det stadig er et Brauntræ. Det gør vi med hjælpefunktionen rebalance. Rebalance kaldes med new→left - i dette tilfælde 2, og dette er nu toppen i det undertræ, som vi rebalancerer. Der fortsættes med rebalancering indtil der findes et undertræ med en højrepeger, som peger på NULL, i hvilket tilfælde vi er færdige med at balancere træet, og vi igen burde have et fint Brauntræ.

2.2 Implementation af remvR

Funktionen remvR skal fjerne det sidste element i træet. Til dette skal vi bruge størrelsen på træet, som vi finder ved hjælp af size, som allerede er implementeret. Derudover skal lookup kaldes, så vi får fat i den data, som vi skal have ud af træet, inden elementet slettes. Til selve sletningen bruges en hjælpefunktion (del), som tager træet selv, og dets størrelse som parametre. Vi skal huske at trække 1 fra størrelsen vi kalder del med, da denne gerne skal pege på det sidste element, og size først stopper når der ikke er flere elementer.

Hjælpefunktionen del står for selve sletningen af elementet. Der er 5 forskellige basistilfælde, som beskrives nedenfor.

- size¡0: Ikke-validt index, og funktionen afsluttes.
- size==0: Træ med et element. Hvis vi sletter det ene element i træet, svarer det til at slette hele træet. Vi kalder free på træets peger, og fjerner denne fra maskinens hukommelse.
- size==1: Træ med to elementer - roden og et venstre barn. Vi vil derfor gerne slette det venstre barn. Dette gøres ved at kalde free på rodens venstre peger, og derefter sætte denne til NULL.
- size==2: Træ med tre elementer - roden og både et højre og et venstre barn. Da Brauntræer skal være binære med evt. overvægt til venstre er det denne gang det højre barn, der skal slettes, og vi kalder derfor free på rodens højre peger, og sætter denne til NULL.
- andet: Hvis der er flere elementer i træet skal vi arbejde os nedad træet i dybden for at finde det sidste element. Dette gør vi ved at bruge funktionen odd(i) for at fortælle os om vi skal bevæge os ned i det venstre eller det højre undertræ. Derefter kaldes del igen med det gældende subtræ med rodens enten højre eller venstre barn som den nye rod.

2.3 Testprogram til ovenstående implemenationer

Brauntest er vores test program til brauntræer.

Det virker ved først at indsætte et antal elementer, derefter gennemgår det hvilke elementer der er i træet og til sidst fjernes elementerne igen.

Programmet kan tage op til 3 valgfrie argumenter. Første argument er antallet af knuder der skal indsættes i træet. Andet argument er hvilken add funktion der skal bruges, l for addL og r for addR. Tredje argument er hvilken remv funktion der skal bruges, l for remvL og r for remvR.